LANE KEEPING AND PEDESTRIAN AVOIDANCE FOR A VISION-BASED AUTONOMOUS TEST VEHICLE

A Thesis

by

FORREST BRADLY BERG

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Won-jong Kim
Committee Members,	Sivakumar Rathinam
	Shankar Bhattacharyya
Head of Department,	Andreas A. Polycarpou

August 2016

Major Subject: Mechanical Engineering

ABSTRACT

This thesis presents an experimental platform for the demonstration and testing of autonomous lane keeping and pedestrian avoidance through the use of a test vehicle equipped with an Xbox Kinect 2.0 for vision-based detection. The test vehicle is a 1/5th-scale electric remote control car customized for autonomous steering and drive motor control through the use of an Arduino Mega 2560. A proportional derivative (PD) steering controller running on the Arduino receives position commands from a laptop via serial communication. These commands are generated on the laptop based on an analysis of images captured from the Kinect. On the laptop, the program Processing is used to identify the colored boundaries of the path the vehicle is traveling, calculate the position of the center of that path, find the position error of the test vehicle relative to the center of the path, and then send that error to the Arduino to calculate a corrective steering command. In addition to lane keeping, the test vehicle is capable of pedestrian detection and avoidance through the use of body tracking libraries written for the Kinect in Processing. Once a human body is detected and tracked, the position of each foot is checked relative to the boundaries of the path. If the pedestrian is located inside of the path boundaries, then the test vehicle stops and waits for the person to leave the path. The PD controller for the steering servo was tuned empirically, and after tuning, the vehicle was

consistently able to autonomously navigate a curved path of variable width with an error of less than 75 pixels (5cm). The pedestrian-avoidance algorithm worked successfully in conjunction with the lane keeping algorithm. However, the body tracking libraries for the Kinect demonstrated a 40% failure rate for body detection when the test vehicle and the Kinect were in motion.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Won-jong Kim for his support when I chose to change thesis topics late in my masters program, and for his advice and guidance throughout the research process. I would like to thank Dr. Bhattacharyya and Dr. Rathinam for participating on my committee and taking the time to attend my defense even though they were traveling. I would also like to thank my parents and family for taking the time to proofread my thesis and provide feedback on how it could be improved. And finally I would like to thank my friends Victor and Kenny for helping me debug my code and perform experiments.

NOMENCLATURE

BLDC	Brushless Direct Current
D	Derivative
DC	Direct Current
ESC	Electronic Speed Controller
e(t)	Error as a function of time
fps	Feet per second
I	Integral
IDE	Integrated Development Environment
IR	Infrared
K_d	Derivative gain
K_i	Integral gain
K_p	Proportional Gain
KV	rpm/Volts
LDW	Lane Departure Warning
LiPo	Lithium Polymer
P	Proportional
PD	Proportional Derivative
PI	Proportional Integral
PID	Proportional Integral Derivative
PWM	Pulse Width Modulation

P10-P90	Probability 10% to Probability 90%
RGB	Red Green Blue
USB	Universal Serial Bus
u(t)	Controller Input

TABLE OF CONTENTS

LIST OF FIGURES

Page

1 INTRODUCTION AND LITERATURE REVIEW

1.1 Thesis Overview

This thesis begins with an introduction and literature review. This first section discusses industry applications for lane keeping and human tracking technologies. Section 2 provides a description of the hardware and software components of the project. In this chapter the components of the test vehicle are broken down in detail, and the software used for device communication is explained. Section 3 discusses the development of the controller equations used for steering control and for drive motor control. Section 4 provides an in depth description of the lane keeping algorithms and pedestrian avoidance algorithms. Section 5 provides an analysis of the steering controller gain-tuning process, analyzes the results of the pedestrian avoidance algorithms and reviews the limitations of the Kinect and how they affect the data gathering process. Section 6 provides conclusions and includes discussion about potential future work. The appendix contains the code used in this project.

1.2 Design Objectives

The objective of this thesis is to develop a small-scale autonomous vehicle capable of clearly identifying the boundaries of the path it is navigating while also searching for and avoiding pedestrians that are in the way. To better describe this goal, the design objectives are listed as a set of priorities:

1. Avoid pedestrians by stopping the vehicle completely until they have moved outside of the boundaries of the path.
2. Keep the vehicle as close to the center of the path as possible.
3. Reach the end of the path and stop.

1.3 Lane-Keeping

Many automotive companies and research laboratories have introduced safety technologies into their vehicles to help reduce the number of accidents by alerting drivers to potential danger [1], [2] [3], [4]. These technologies include Lane Departure Warning (LDW) and Lane Keep Assist (LKA). In 2014, General Motors introduced both LDW and LKA to the 2015 model Cadillacs such as the 2015 ATS, 2015 CTS, and 2015 XTS [5]. These systems use image-processing software to identify unintentional lane departures, alert the driver, and help direct the vehicle back towards the center of the lane. The LDW system warns the driver through a warning sound or vibration in the seat. The LKA system applies a small torque to the steering wheel to prevent an unintentional lane departure [5]. Similar technologies have been developed by Ford Motor Company [6], Toyota [7], Lincoln Motor Company [8], and Google [9]. Tesla is working to implement an autopilot system that would enable their vehicles to perform hands-free lane keeping among other safety features [10].

1.4 Human-Tracking

Human-tracking technology has recently become accessible to the general public thanks to affordable devices such as the Xbox Kinect from Microsoft [11] and the Xtion PRO Live from ASUS [12]. These devices have robust human-tracking capabilities that can be used by developers for many applications. These human-tracking sensors function by processing information from a depth map and color image in order to distinguish human bodies from their surroundings [13]. At Microsoft Research Cambridge & Xbox Incubation, Shotton et. al. developed human-tracking algorithms that are able to identify human bodies of varying proportions in a large number of different poses just from a single-depth image[14]. Their highly

accurate body-tracking algorithms function by utilizing a large database of various body poses that are compared against the pose of the human body in the view frame. Hand and gesture tracking is precise enough using the Kinect that Li was able to use it to develop an algorithm to recognize and translate sign language [15]. Human-tracking technologies have a wide variety of applications, some of which will be discussed in the following section.

1.5 Applications

Lane-keeping and human-tracking technologies have applications in a variety of technologies, including autonomous vehicles and robotic delivery systems.

1.5.1 Automotive

Lane keeping and human-tracking technologies have obvious safety applications in the automotive industry. As active safety systems such as LKA, LDW, cruise control, anti-lock braking, traction control, and pedestrian and obstacle avoidance improve, vehicles so equipped will require much less input from the driver and will decrease the number of vehicle collisions around the world [16], [17]. A 2015 National Highway Traffic Safety Administration report showed that 94% of the approximately 2.2 million automobile accidents in the United States occurred because of human error [18]. The National Highway Traffic Safety Administration and the Insurance Institute for Highway Safety recently announced the commitment of 20 automakers to implement autonomous emergency breaking as a standard feature for all new cars by 2022. The automakers involved, Audi, BMW, FCA US LLC, Ford, General Motors, Honda, Hyundai, Jaguar, Land Rover, Kia, Maserati, Mazda, Mercedes-Benz, Mitsubishi Motors, Nissan, Porsche, Subaru, Tesla Motors Inc., Toyota, Volkswagen and Volvo Car USA, represent 99% of US vehicle manufacturing [19]. Many of these prominent automotive companies intend to take

safety even farther by eventually implementing fully autonomous control of their vehicles [9], [20], [21], [22]. Google's self-driving vehicles are capable of highly accurate lane keeping and pedestrian avoidance and no longer require any input from a driver at all in order to function properly, and these vehicles have driven more than 1.5 million miles with minimal accidents. To emphasize its autonomous nature, a steering wheel is not even included in their test vehicles [9].

1.5.2 Product Delivery

Online retailer Amazon uses Kiva robots in their warehouses to aid with the selection and transport of goods to be mailed to consumers[23]. These robots essentially automate the entire picking and packing process for Amazon's large warehouses. The robots are capable of navigating throughout a warehouse to find specific shelves full of products, and then delivering these products to an employee that is then able to pack them up for shipping. These robots are able to navigate through complex paths without colliding with any products or with each other by using complex navigation and lane-keeping algorithms.

Another type of product delivery system that uses human tracking and lane-keeping technologies is the TUG delivery system designed by Aethon for use in hospitals[24]. The TUG robot is basically a mobile cabinet that can be programmed to bring various medical supplies, meals for patients, clothing, and more to any room in a hospital. The TUG robots use lane-keeping to navigate through hallways and also employ object detection and human tracking capabilities in order to avoid collisions.

1.6 Thesis Contributions

This thesis presents a lane-keeping and pedestrian-avoidance system using low cost hardware available to the general public. Potential applications of this technology include robotic delivery systems in warehouses or other industrial environments, and potentially in locations with pedestrian traffic. This thesis also tests algorithms that, with more robust hardware, might have applicability for automobile pedestrian-avoidance.

2 DESIGN AND SYSTEM ARCHITECTURE

This section describes the hardware and software used in this thesis. It begins with a description of the Xbox Kinect 2.0 sensor shown in Figure 1, the test vehicle and the Arduino Mega 2560, and then moves on to describe the Processing and Arduino integrated development environment (IDE), followed by an explanation of device communication and connections.

2.1 Hardware Components

2.1.1 Kinect Sensor

Figure 1: Photo of Kinect sensor

The position of the vehicle relative to the boundaries of the path it was following and relative to the location of pedestrians on the path were calculated from information gained through the use of an Xbox Kinect 2.0 sensor system shown above. The Kinect has a color camera capable of capturing 1920 $\times$ 1080 resolution images at 30 frames per second (fps). Shown in Figure 2, the Kinect utilizes three infrared

(IR) light emitters and a time of flight camera to create a 3D map of everything within its view range. The time of flight camera calculates depth by measuring the amount of time it takes for the IR light to bounce off of objects and return to the camera. This allows for accurate depth sensing capabilities that can be used to identify and track the motion of human bodies.

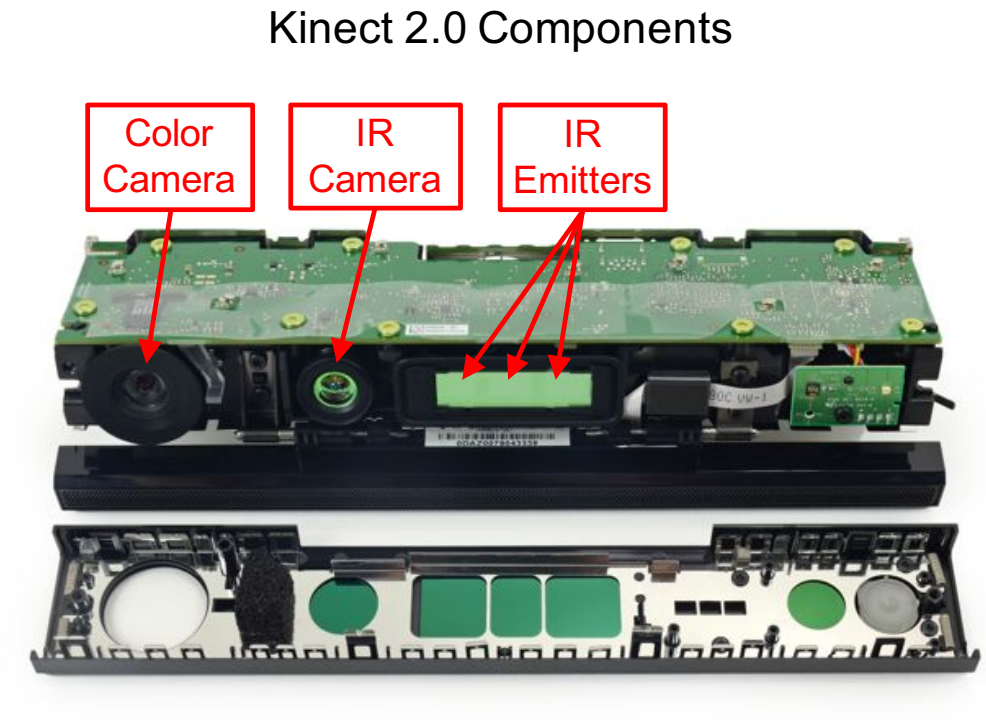

Figure 2: Kinect 2.0 components [25]

The depth sensing camera has a resolution of 512 $\times$ 424, a maximum working distance of approximately 4.5 m, and a minimum working distance of 50 cm. The horizontal field of view is 70°, and the vertical field of view is 60°. This limited field of view was the cause of some problems during testing of the pedestrian avoid-

ance algorithms that will be discussed in Section 4. The color camera was used to identify and track the boundaries of the path the vehicle traveled. The boundaries of the path were marked using unique colors that were easily recognized and distinguished from other objects in the field of view. The Kinect 2.0 is also capable of tracking up to 6 people simultaneously and identifying up to 26 joints for each human body. This joint tracking capability was adapted to identify the position of a pedestrian relative to the test track [26].

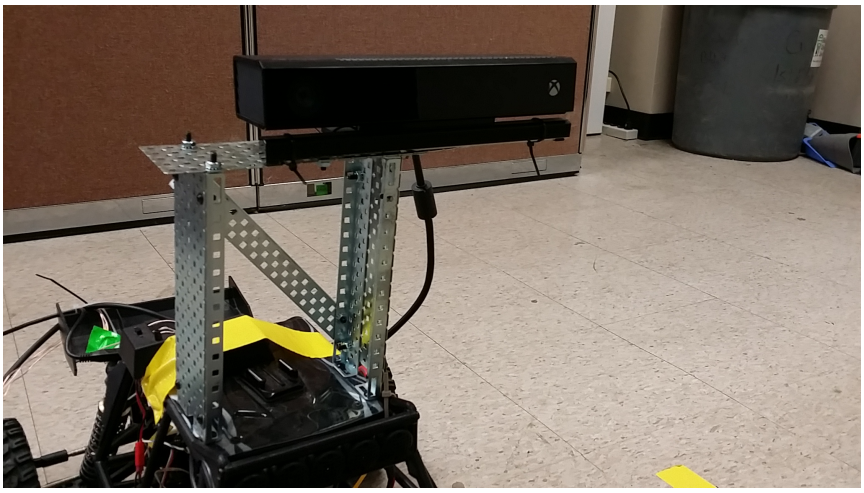

Figure 3: Aluminum tower used to mount the Kinect on the test vehicle

During testing, the Kinect was mounted to the test vehicle using a small tower constructed from aluminum plating shown in Figure 3. This mounting system was designed to elevate the Kinect to make it easier for the body tracking software to function properly. The Kinect was also mounted at an offset from the center of the test vehicle because the color camera, located on the far right end of the device, needed to be centered on the vehicle in order to ensure that the field of view was

centered on the direction that the vehicle was traveling.

2.1.2 Test Vehicle

The test vehicle used for this project (shown in Figure 4) was a Rampage Chimera EP Pro, a 1/5th-scale 4-wheel drive model car powered by a 980KV Brushless Direct Current (BLDC) motor [27]. The chassis was made of 4 mm thick Aluminum 6061T6, and the wheelbase is 500 mm [28]. The BLDC motor is controlled by an Electronic Speed Controller (ESC) and is powered by a 22.2 V 6-cell Lithium Polymer (LiPo) battery shown in Figure 5.

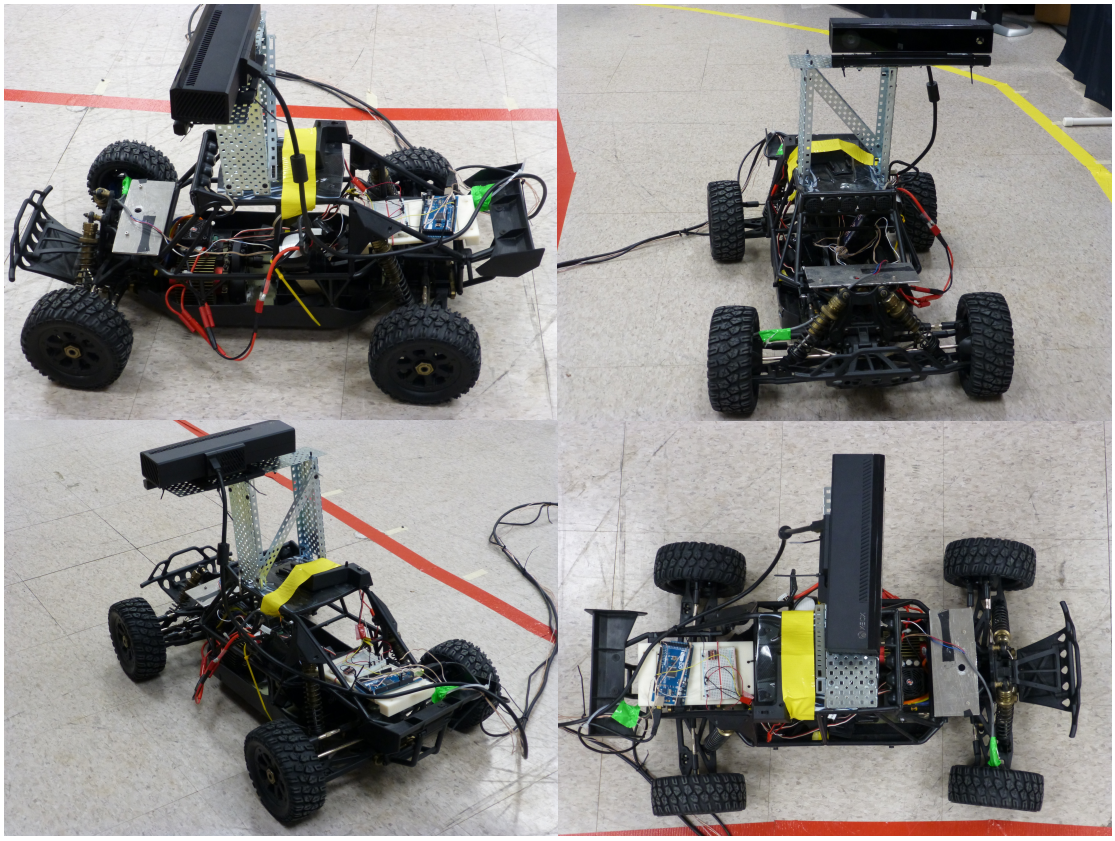

Figure 4: Test vehicle photos

The ESC simplifies the control of the motor by using Pulse Width Modulation (PWM) to control the magnitude of the output, which is the same method used to control the steering servos. The ESC is capable of handling up to 150 A and includes multiple safety features such as low-voltage cut-off protection, over-heat protection, and throttle-signal-loss protection. It also has 3 running modes, 9 start modes, and 4 levels of brake power adjustment. For the experiments discussed in this paper, the ESC was configured so that the vehicle was only capable of moving in a forward direction, with speed controlled by the magnitude of the PWM signal sent from the Arduino microcontroller.

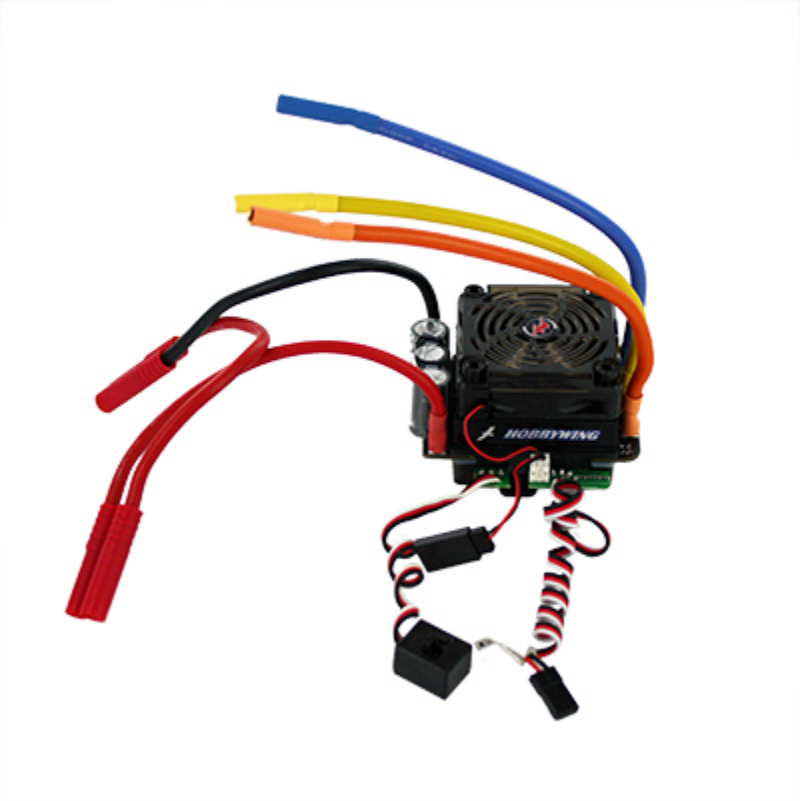

Figure 5: Electronic speed controller

2.1.3 Arduino Mega 2560

The Arduino Mega 2560 microcontroller board, shown in Figure 6, was designed around the ATmega2560 microcontroller. The board comes with 54 digital i/o pins, 16 analog input pins, and 4 hardware serial ports. Fifteen of the digital i/o pins are capable of PWM. The Mega has an operating voltage of 5 V, but has a single 3.3 V DC current pin capable of supplying 50 mA of current that can be used for interfacing with lower voltage devices. It has a clock speed of 16 MHz and has 256 kilobytes of flash memory [29]. This board was chosen because of how easily accessible the PWM pins and hardware serial ports are, and its high clock speed. Additionally, libraries have been developed for the Arduino Integrated Development Environment (IDE) that simplify servo control and information transfer.

During operation, the desired steering angle was sent to the Arduino Mega from a laptop, enabling the Arduino to command the steering servos to the desired angle through the use of PWM. The algorithms used to calculate the desired steering angle are discussed in Section 4.

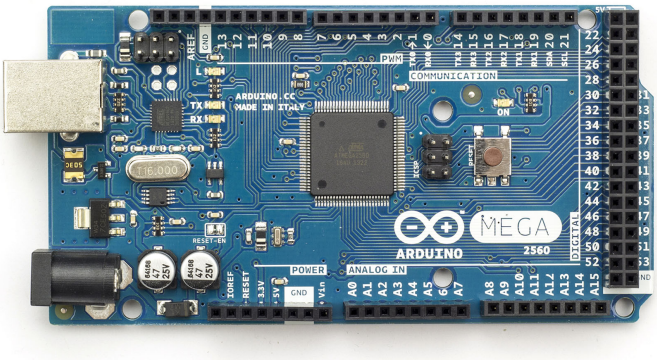

Figure 6: Arduino Mega 2560

2.2 Software Components

2.2.1 Arduino IDE

The Arduino IDE is designed for use with C/C++ programming languages and has a wide variety of libraries for many common robotics applications. One of the major benefits of working with Arduino products is the large community that provides advice and troubleshooting support on the forums of the Arduino website. Functions from the servo library and serial communication library were used for this project in order to control both the steering and drive motors through PWM signals, and to receive data through the hardware serial ports.

2.2.2 Processing IDE and Kinect 2.0 Library

The Processing IDE is designed with visual arts in mind and has many functions and libraries that allow for the manipulation of pixels. The Kinect 2.0 Library, written by Lengeling, has functions that allow a user to access and manipulate the color and depth pixel data from the Kinect [30]. These libraries also allow a user to enable the Kinect's various body tracking capabilities. The Kinect 2.0, library in conjunction with the Processing IDE, provided all of the functionality needed for the development of the pedestrian avoidance algorithms discussed in Section 4.

2.2.3 Device Communication

Realization of the project's design objectives required the ability to strictly control the angle of the front wheels of the vehicle, the amount of current sent to the drive motor, and real time body tracking for pedestrian avoidance. The PWM signals for steering control were generated based on an analysis of the images sent from the Kinect to a laptop. The Processing IDE provided capabilities that were used to facilitate data transfer between the Kinect and the Arduino. By using Kinect

libraries written for the Processing IDE, it was possible to develop a algorithm that could identify the boundaries of the road that the test vehicle was traveling on, navigate to the center of the road, and detect and avoid pedestrians that were blocking the path of the vehicle. The position error of the car relative to the center of the road was sent from the laptop to the Arduino Mega using serial communication. The PWM values for steering angle were generated on the Arduino Mega by a PD controller and then sent directly to the steering servos through the use of the Arduino servo library.

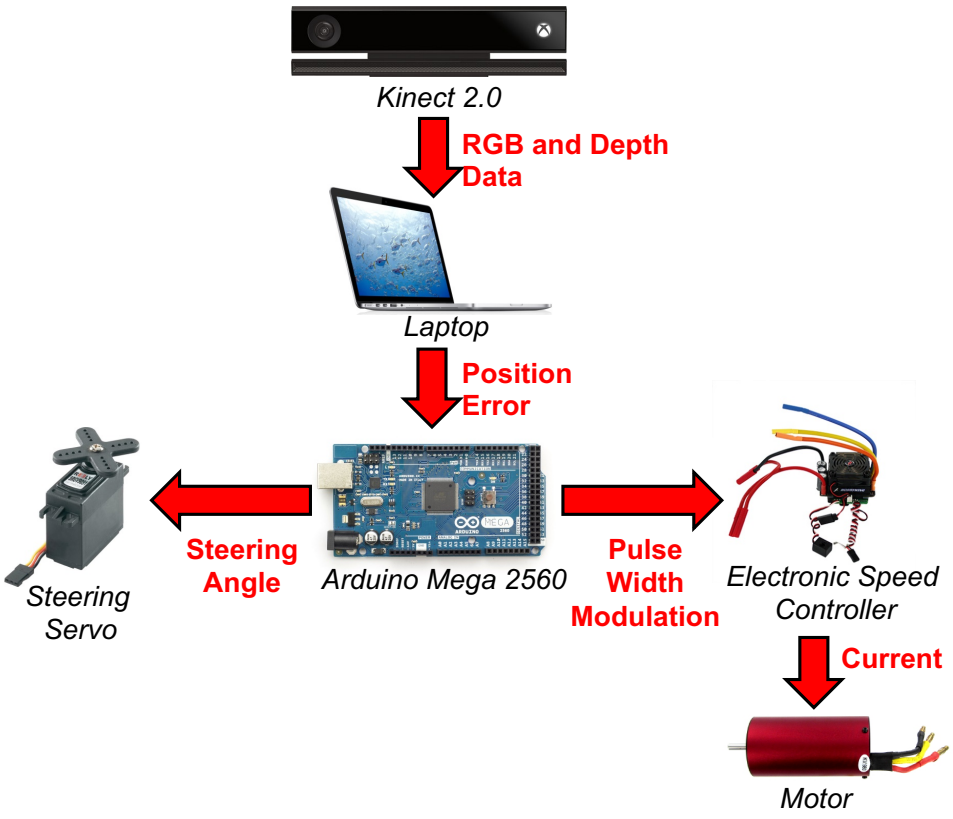

Figure 7: Block diagram showing device communication path

The amount of current sent to the BLDC motor was controlled by PWM signals sent

to the ESC that was described previously. These PWM signals were generated using an on-off controller that analyzed skeleton tracking data to decide when it was safe for the vehicle to move down the road without risking collision with a pedestrian. Figure 7 illustrates the flow of information from the Kinect to the BLDC motor and the servos that control the steering angle.

2.2.4 Serial Communication Using Processing

The KinectPV2 library written for Processing by Lengeling uses the Processing IDE to simplify data access from the Kinect [30]. As discussed previously, the library has functions for performing body tracking of up to six people and is able to identify and map the coordinate data for each of their joints. Color and depth images are sent to a laptop from the Kinect through a USB 3.0 using serial communication at a rate of 30 fps. This high rate of data transfer was the reason that it was not possible to perform experiments wirelessly, which therefore required that the vehicle be tethered to the computer with the USB 3.0 cable. In the Processing IDE, there is a library designed for serial communication using the USB ports on the laptop. This functionality was utilized to send the reference values for torque and steering angle from the laptop to the Arduino Mega. Algorithm 1 demonstrates how the serial communication library was used.

Algorithm 1 Serial communication from Processing to Arduino Mega

```
1: import processing.serial.*; // Import processing serial library
2: String portName = Serial.list()[0]; // create a serial port object for the correct com port
3: myPort = new Serial(this, portName, 9600); // select the correct com port and set the BAUD rate
4: calculate position error
5: myPort.write(xError); // send the position error to the Arduino via the selected serial Port
```

The methods used for calculating the position error from the color and depth images will be discussed in Section 4.

2.2.5 Arduino Mega Serial Communication

The serial capabilities of the Arduino Mega were used to receive the position error data from processing at a baud rate of 9600. A description of the algorithm that was run on the Mega is shown in Algorithm 2.

Algorithm 2 Arduino Mega servo control and data transfer

```
#include <Servo.h> // import Servo library
Serial.begin(9600); // Start serial communication at 9600 bps in the setup function
Servo drive; // create a servo object for the drive motor
Servo steering; // create a servo object for the steering servo
drive.attach(8); // attach the drive motor to pin 8
steering.attach(9); // attach the steering servo to pin 9
loop
    while Serial.available() == 0 do
        // Do nothing while waiting for position error data to be sent from the Processing function
    end while
    posError = Serial.read(); // receive the data and store it in a variable
    calculate desired steering angle with PD controller
    calculate desired drive motor PWM signal with on off controller
    steering.write(steeringAngle); // send desired steering angle to the steering servo
    drive.write(drivePWM); // send the desired PWM signal to the drive motor
end loop
```

2.3 Device Connections

As discussed previously, all device connections were hardwired. Although it would have been convenient, it was not possible to perform any experiments wirelessly due to the volume of data being transferred from the Kinect to the laptop. The Kinect transfered data to the laptop using a USB 3.0 connection, which was necessary to transfer 30 fps of high resolution color images. The Arduino Mega was connected to the laptop via a USB 2.0 cable for the transfer of position error data

from the laptop's serial ports to the serial port on the Arduino Mega. Figure 8 shows how the Arduino mega was connected to the ESC for drive motor control, to the steering servos, and to an emergency stop switch. The connectors to the ESC and steering servo have slots for a ground-cable, a power-cable, and for a PWM signal-cable. The ESC was connected to a 22.2 V LiPo battery through a separate connector so only the signal and ground cables needed to be connected to the Arduino. These connections were secured with yellow tape as shown in Figure 8.

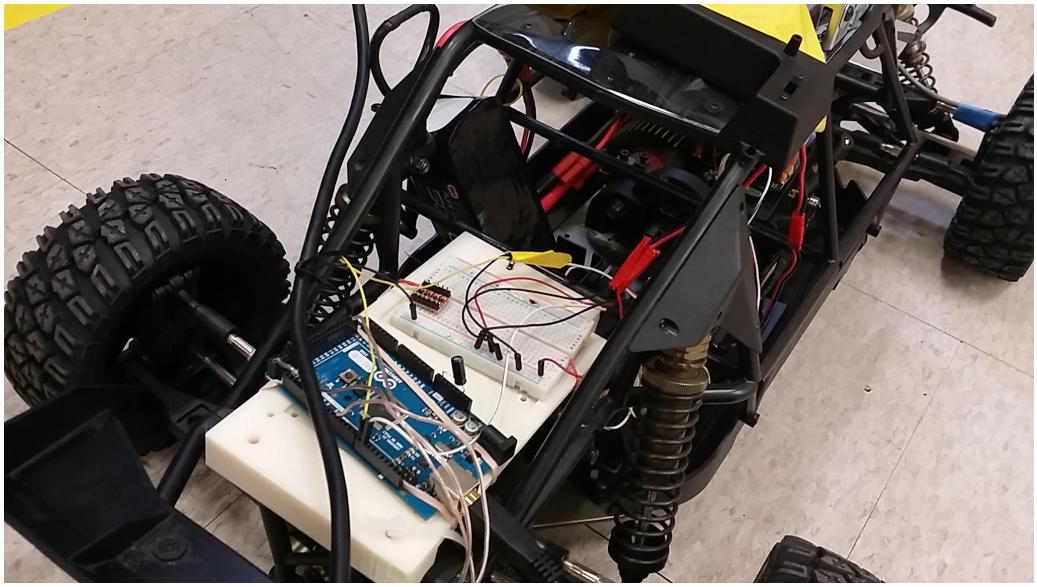

Figure 8: This figure shows the mount used for the Arduino and the connections between the Arduino, the ESC, and the steering servo

The steering servo connection required a separate 6V battery pack, and both the servo and 6 V battery had to be connected to the Arduino ground. To form this circuit safely, it was necessary to use a small breadboard. Using the breadboard,

it was simple to connect the ESC and the steering servo to an Arduino ground pin. The battery for the steering servo was wired through an external on/off switch that provided capability to easily disable power during operation. For extra safety during testing, an emergency stop switch was built and attached to an external interrupt pin on the Arduino Mega. This required 3 wires: one to the 5 V pin, one to a ground pin, and one attached to a digital pin. When the emergency stop switch was toggled off, the Arduino recognized a change in voltage and disabled the drive motor.

3 CONTROLLER DEVELOPMENT

The test vehicle had two degrees of freedom (DOFs), each of which required controllers. The first and most complex of the two was a PD controller for the steering servos. The second was on-off control of the drive motor. This chapter will discuss why each type of controller was chosen and how it was tuned.

3.1 PID Control

Proportional Integral Derivative (PID) Control is a robust control method that has a wide range of applications [31]. Each of the three components of the controller work together to minimize error in their own way. The proportional (P) term, shown in (1), changes the controller output by a factor directly proportional to the error, e(t).

$$u(t) = K_p e(t) \tag{1}$$

The parameter K_P is called the proportional gain and is tuned in order to adjust the magnitude of the response of the controller. When tuning a PID controller empirically, the P gain is increased until the system responds with small steady-state oscillations around the set point. Obviously, continuous oscillations around the set point are not optimal, and that is where the remaining terms of the controller come in. The integral (I) term, shown in (2), is designed to adjust the controller output based on the integral of the error over the time the controller has been operating.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau \tag{2}$$

The I term represents the sum of all of the previous error values over time multiplied

by integral gain K_i. This term is designed to minimize steady state error. When combined with the P term it creates a controller that is capable of minimizing error around a constant set point. A controller that combines the P term and I term is called a PI controller and is one of the most commonly used combination of (3) adds the derivative term to the controller.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{3}$$

The time derivative of the error is calculated by storing the current and previous values of error and time and is shown in equation (4).

$$\frac{de(t)}{dt} = \frac{(current\ error - previous\ error)}{(current\ time - previous\ time)} \tag{4}$$

When the D term is added, the controller usually is able to provide damping to the controller output that can be optimized by tuning the value of K_d.

For this thesis, a PD controller was chosen to provide steering control rather than a PID controller. Frequently, PD control is more effective for servo applications because it trades minimal steady-state error response for fast response times. The steering controller used for this project does not require the steady-state error be minimized to near 0 for the car to safely and efficiently navigate the path it is on. The reference value for this controller was changing rapidly in the lane keeping experiments, so it was necessary to prioritize fast response time over minimizing steady state error [32]. The final controller equation used for this thesis is shown in (5).

$$u(t) = K_p e(t) + K_d \frac{de(t)}{dt} \tag{5}$$

Figure 9 shows the block diagram for the steering PD controller, describing the

finalized control loop that ran on the Arduino.

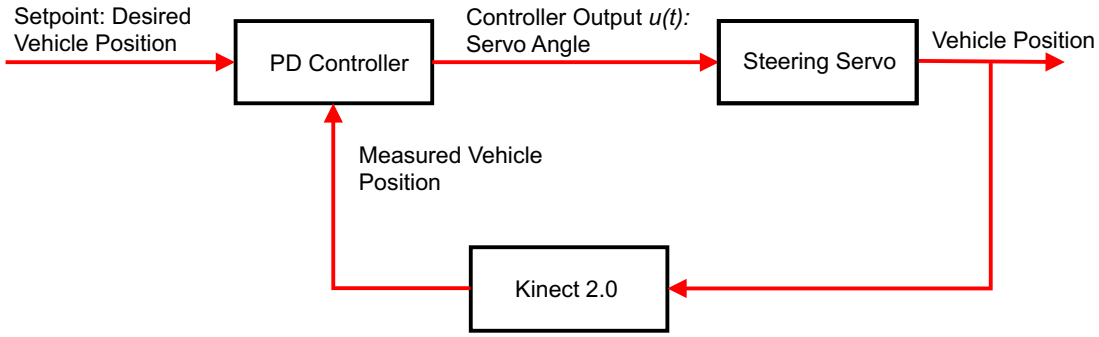

Figure 9: Block diagram

The PD controller receives position error which was calculated by subtracting the current vehicle position from the desired position. The controller then generates an output, u(t), which is sent to the steering servo. As the car moves, the Kinect measures the new position and feeds that information back into the PD controller to begin the loop again.

3.2 On-Off Control

An on-off controller was chosen for the drive motor because it was robust enough to achieve the design objectives. This controller is primarily designed around the pedestrian avoidance algorithm discussed in Section 4. The primary working principle for this controller was to send a constant PWM signal to the ESC that would actuate the motor at a low velocity until either a pedestrian was detected or the end of the path was reached. When a pedestrian was detected as blocking the path, the on-off controller changed the drive PWM signal to a value that stopped the motor, thereby braking the vehicle, until the pedestrian exited the path. Section 6 discusses some new, additional, design objectives that would require a more

robust controller to allow for precise velocity control.

4 LANE KEEPING AND PEDESTRIAN AVOIDANCE

This section explains the methods and algorithms used to achieve lane keeping and pedestrian avoidance. Section 4.1 describes how lane keeping is achieved, and Section 4.2 shows how pedestrian avoidance is incorporated into the lane-keeping algorithms. Figure 10 shows how the algorithms worked together to achieve the design objectives.

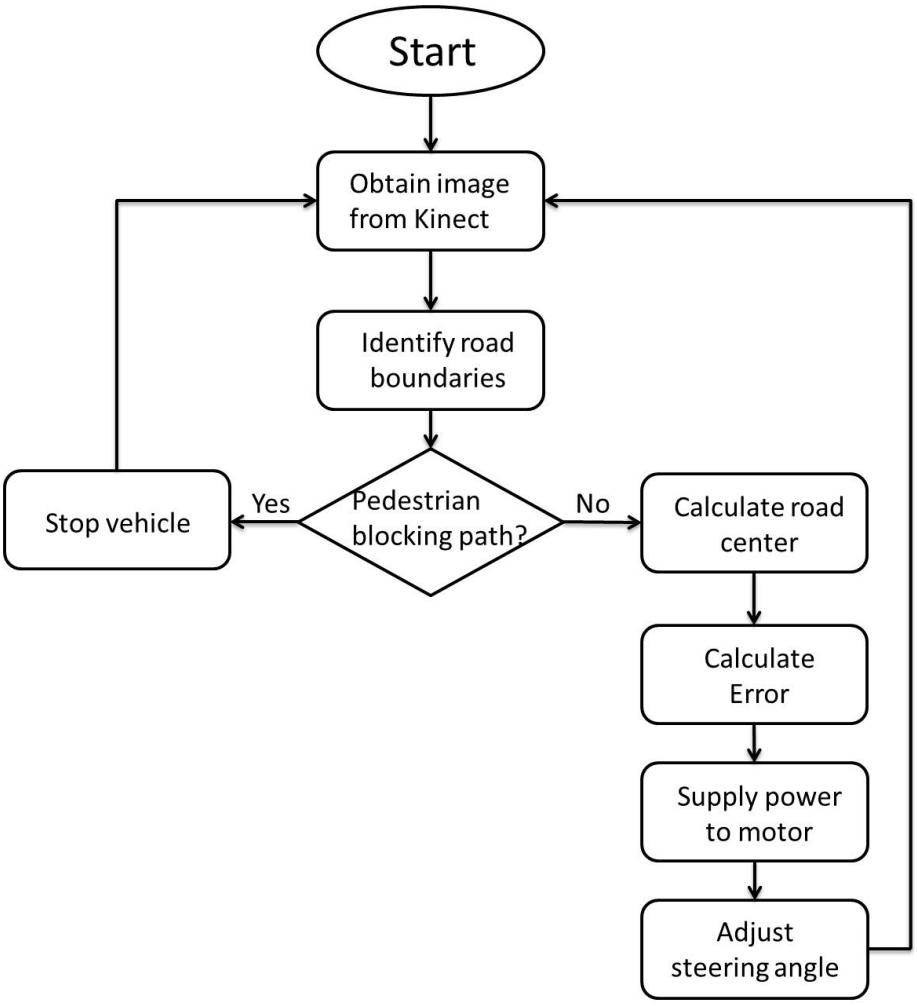

Figure 10: Algorithm flow chart

4.1 Lane-Keeping

The lane keeping algorithms were written using the Processing IDE and made use of the KinectPV2 library. A high level description is shown in Algorithm 3.

Algorithm 3 Path Navigation and Lane Keeping

1: Initialize color tracking library.
2: Identify the two boundaries of the path based on their unique color.
3: Find the coordinates of the left and right path boundaries at a fixed distance in front of the vehicle.
4: Calculate the position of the center of the path based on the coordinates of the path boundaries.
5: Calculate the error of the vehicles position relative to the center of the path.
6: Calculate a new steering angle that will drive this error to 0.
7: Send the steering angle to the Arduino.
8: Repeat.

4.1.1 Path Boundary Identification with Color

As discussed previously, the Kinect is capable of gathering 1920×1080 resolution color images at a rate of 30 fps. It is capable of identifying a red, green, and blue (RGB) value ranging from 0 to 255 for each individual pixel through the use of built-in functions available in the Processing IDE. The Kinect was used to identify the boundaries of the path that the car would follow by finding the pixels with the desired RGB values. The colors red and yellow were chosen to create the boundaries of the path because the Kinect was most consistently able to distinguish them from the background colors during testing.

Figure 11 shows an unprocessed image of the test path boundaries as generated by the Kinect.

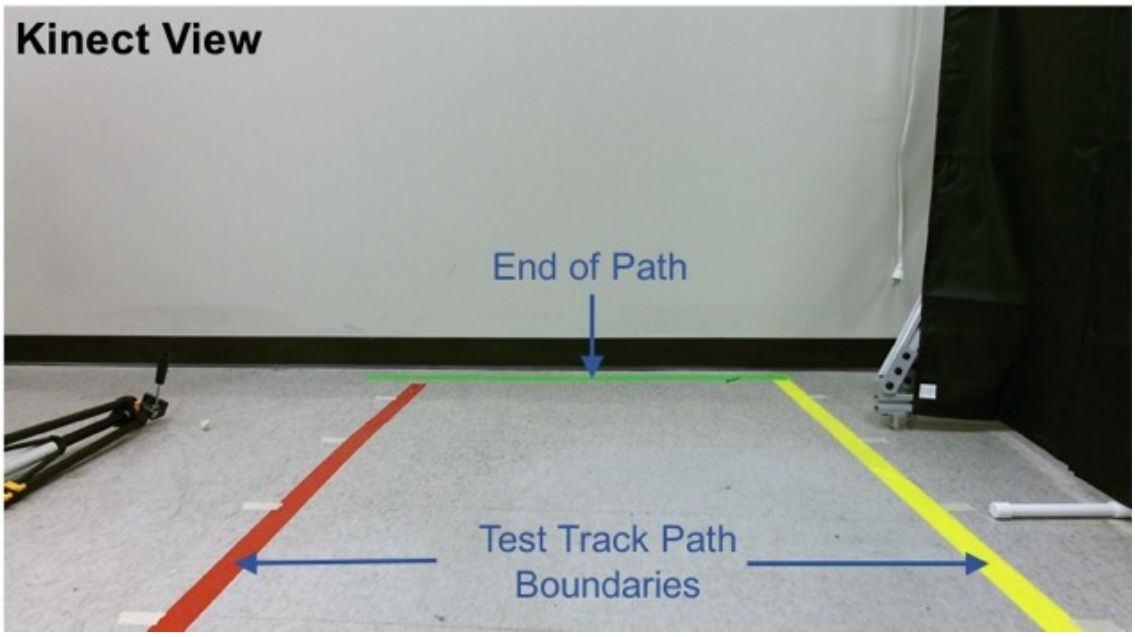

Figure 11: Kinect color image showing the untracked path boundaries of the test track

In the experiment, the red and yellow path boundaries were identified by their RBG values, allowing the boundary pixels to be grouped into two arrays, one for each color. Isolating the set of pixels with RGB values that corresponded with each boundary was done through the use of nested for loops and through a relative evaluation of RGB values.

Algorithm 4 Identification of path boundaries using pixel color

1: For(int x = 0; x $<$ 1920; x++)
2: For(int y = 0; y $<$ 1080; y++)
3: color pix = get(x,y) // designates the current pixel as a color object
4: if(abs(green(pix) - red(pix)) $\leq$ 40 & abs(red(pix) - blue(pix)) $\geq$ 70 & abs(green(pix) - blue(pix)) $\geq$ 70) then store pix in yellow array // pixel color differences used to identify yellow tape
5: if(abs(red(pix) - green(pix)) $\geq$ 60 & abs(red(pix) - blue(pix)) $\geq$ 50 & abs(blue(pix) - green(pix)) $\leq$ 30) then store pixel in red array // pixel color differences used to identify red tape

As discussed, the Kinect assigns each pixel with a red, green, and blue value ranging from 0 to 255. The “if” statements in the algorithm show how pixel RGB values were compared in order to determine if they needed to be stored, and if so, which array to store them in. Algorithm 4 stores all the pixels within the strips of tape used to represent the boundaries of the path.

Once all of the pixels in the boundaries of the path were identified and stored, it was possible to further optimize this algorithm by storing only the pixels that represented the inside edges of the tape representing the path boundaries. Storing only the edge pixels drastically reduced the size of the storage arrays which increased the processing speed for the program. Algorithm 5 improves on Algorithm 4 by isolating the pixels on the inside edges of the tape.

Algorithm 5 Optimization of algorithm 4 by only storing pixels that are located on the inside edges of the path boundaries

```
1: prevPix = get(x-1,y); // pixel to the left of current pixel
2: nextPix = get(x+1,y); // pixel to the right of current pixel
3: For(int x = 0; x < 1920; x++)
4: For(int y = 0; y < 1080; y++)
5: if(get(x,y) is yellow) store in yellow array
6: // the variable groundTolerance represents the color value used to identify the white tile floor that experiments were run on
7: if(abs(green(prevPix) - red(prevPix)) ≤ groundTolerance & abs(green(prevPix) - blue(prevPix)) ≤ groundTolerance & abs(blue(prevPix) - red(prevPix)) ≤ groundTolerance) then story x and y values in yellow arrays
8: if(get(x,y) is red) store (x,y) in red array
9: if(abs(green(nextPix) - red(nextPix)) ≤ groundTolerance & abs(green(nextPix) - blue(nextPix)) ≤ groundTolerance & abs(blue(nextPix) - red(nextPix)) ≤ groundTolerance) then store x and y values in red arrays
```

Algorithm 5 was written to identify the inside boundaries of the tape by comparing the color of each pixel to the pixel on the immediate left or right depending on which lane was being analyzed. Each pixel in the left lane was compared to the pixel to

its immediate right. If the pixel on the right was found to be the color of the tile floor, then the the adjacent lane pixel was identified as the inside edge of the lane, and the coordinates of that pixel were stored into the array for that lane. The same logic was applied to the right lane to identify and store the leftmost pixel of that lane. Figure 12 provides a Kinect generated visual representation of the results of Algorithm 5.

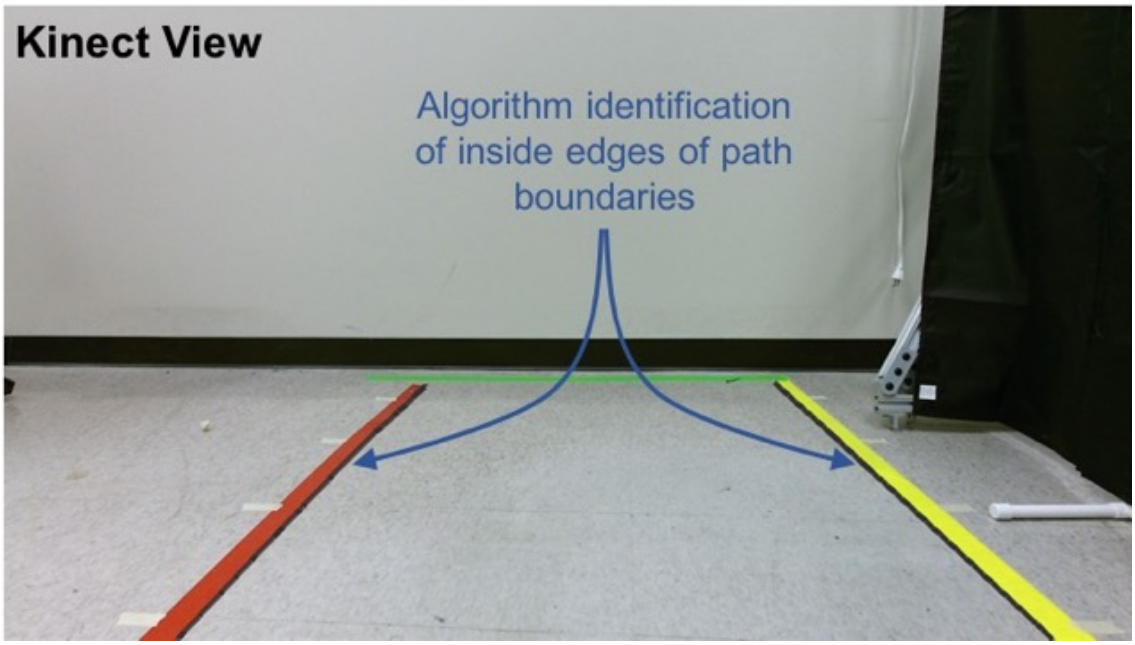

Figure 12: Path boundary identification using algorithms 4 and 5

The black lines on the inner edges of each of the two colored lanes were drawn using the Processing IDE's line function. Lines were drawn by connecting each point stored in the boundary arrays for each color. This visual representation helped to identify how well the boundary identification algorithm was functioning during

testing.

The next step of the process was to dynamically calculate a reference point at a fixed distance away from the car that could be used as a reference value for the steering-angle controller. The math used to calculate this reference point can be seen in Algorithm 6.

Algorithm 6 Finding the position error

1: for(int i = 0; i < yellowBoundaryArray length; i++;) if(yellowBoundaryArray y value == yCenter) then store yellowBoundary Array x and y values at i as centerYellowBoundaryX and centerYellowBoundaryY
2: for(int i = 0; i < redBoundaryArray length; i++;) if(redBoundaryArray y value == yCenter) then store redBoundary Array x and y values at i as centerRedBoundaryX and centerRedBoundaryY
3: roadWidth = centerYellowBoundaryX - centerRedBoundaryX
4: roadCenter = roadWidth/2 + centerRedBoundaryX;
5: posError = xCenter - roadCenter;

Algorithm 6 searched through the arrays containing the coordinates of pixels of the inner path boundaries to find the locations of the boundary pixels to the immediate left and right of the designated lane keeping target (located approximately 0.5 meters in front of the test vehicle). These points were used to calculate the road width and center position in real time. This method was robust enough to allow the car to find the center point of complex winding paths of variable widths. Once the reference point (center of the path) was calculated, it was possible to develop the steering angle controller around that reference. Figure 13 is an image from the Kinect showing the error relative to the reference point calculated by the steering-angle controller. This representation was generated for each time step (approximately 200 ms each) during testing and provided feedback for gain tuning and debugging.

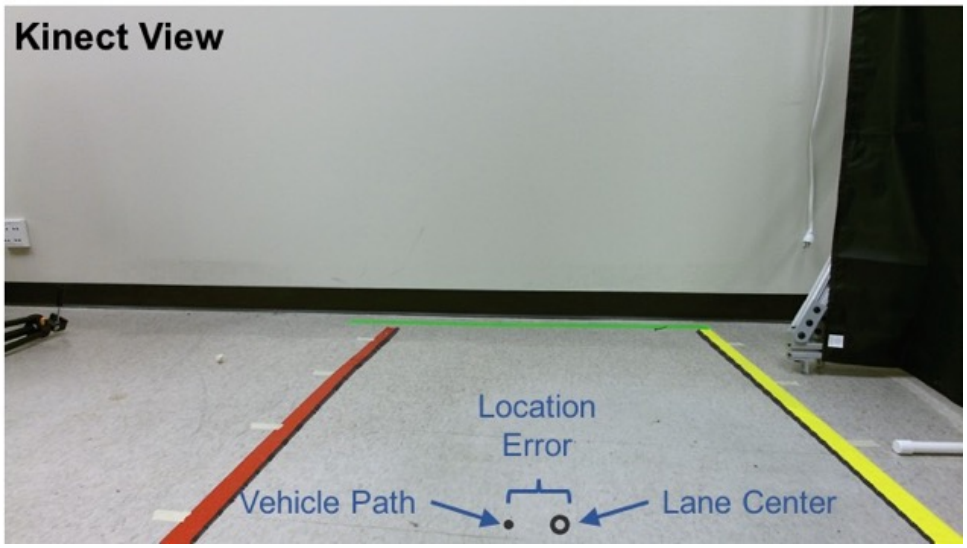

Figure 13: Kinect image showing steering controller error visualization

4.1.2 Steering Controller

As discussed in Section 3.1, a PD controller was used to generate the steering angle for the test vehicle. The optimal values for K_p and K_d were determined through experimentation. The gain tuning process and optimization of the PD controller is discussed at length in section 5. The PD controller was written in the Arduino IDE, and required that the value of the position error be sent via serial communication. The method used for data transfer from the Processing IDE to the Arduino were shown previously in Algorithm 2. An on-off controller was used to control the drive motor. Algorithm 7 shows how the PD controller and on-off controller were developed in the Arduino IDE.

Algorithm 7 This algorithm shows the development of the PD controller for the steering angle and the on-off controller for the drive motor

```
1: receive posError from Processing function // this method is described in Algorithm 2
2: receive directionVariable // Serial communication function can only send positive integers so a switch statement was used to determine the sign of the position error
3: drivePWM = minVelocity;
4: Switch(directionVariable)
5: Case 1: // case 1 entered when directionVariable indicates negative position error
6:     posError = posError*-1; // make error negative
7:     // check to see if the vehicle is starting from a stopped position. If yes, then drivePWM is increased for a preset interval of 200 milliseconds to overcome static friction and initiate motion
8: if startCheck == 0 then
9:     drive.write(startVelocity);
10:    delay(200); // 200 millisecond delay referenced in prior statement
11:    startCheck = 1;
12: end if
13: break;
14: Case 2: // case 2 entered when directionVariable indicates positive position error
15: if startCheck == 0 then
16:    drive.write(startVelocity); // checks to see if the vehicle is starting from a stopped position
17:    delay(200);
18:    startCheck = 1;
19: end if
20: break;
21: Case 3: // case 3 is entered when directionVariable indicates that a pedestrian is detected or the end of the path is reached
22:    drivePWM = throttleNeutral; // stops the vehicle
23:    startCheck = 0; // reset the start up sequence
24: break;
25: Default Case: // if directionVariable is not 1 2 or 3 then stop the vehicle
26:    drivePWM = throttleNeutral;
27: end switch statement
28: currentTime = millis(); // Stores the current time in milliseconds
29: dTime = currentTime - prevTime; // stores the change in time between loops
30: posDError = (posError - prevPosError)/dTime; // stores the derivative of the error
31: angle = P*posError + D*posDError; // PD control on the angle // the following if statements make sure that the value of angle stays within the operating range of the vehicle
32: if angle < -90 then
33:    angle = -90;
34: end if
35: if angle > 90 then
36:    angle = 90;
37: end if
38: steeringAngle = neutralPosition + angle; // the variable angle represents the deviation from the neutral position of 90 degrees
39: steering.write(steeringAngle); // send angle to steering servos
40: prevPosError = posError;
41: prevTime = currentTime;
42: if (estop == 1) drivePWM = throttleNeutral; // emergency stop triggered by switch
43: drive.write(drivePWM); // send drivePWM to the drive motor
```

The steering controller calculates the required steering angle deviation from the neutral position based on the magnitude and the sign of the position error. If position error is negative, the vehicle is steered left, and if the position error is positive,

the vehicle is steered right. The sign of the position error was managed through the use of a switch statement. It was mentioned in Algorithm 7 that Processing can only send positive integers via serial communication, therefore another variable was sent after the position error to designate if the value of position error should be positive or negative. The switch statement was also used to stop the vehicle in the event that the Kinect detects a pedestrian or sees that the test vehicle has reached the end of the path. The steering angle command being sent from the Arduino Mega to the servo motor could only accept values in the range of $0^{\circ}$to 180°, so lines 32-37 of the Algorithm 7 ensure that the calculated value of the steering angle was restricted to the operating range of the vehicle before it was sent from Processing to the Arduino.

4.1.3 Human Tracking with Kinect Sensor

Kinect PV2 library functions were used to determine the location of pedestrians, determine whether they were blocking the path that the vehicle was attempting to navigate and determine whether it was necessary to stop the vehicle to avoid a collision. This library provides a visual representation of the human body tracking shown in Figure 14.

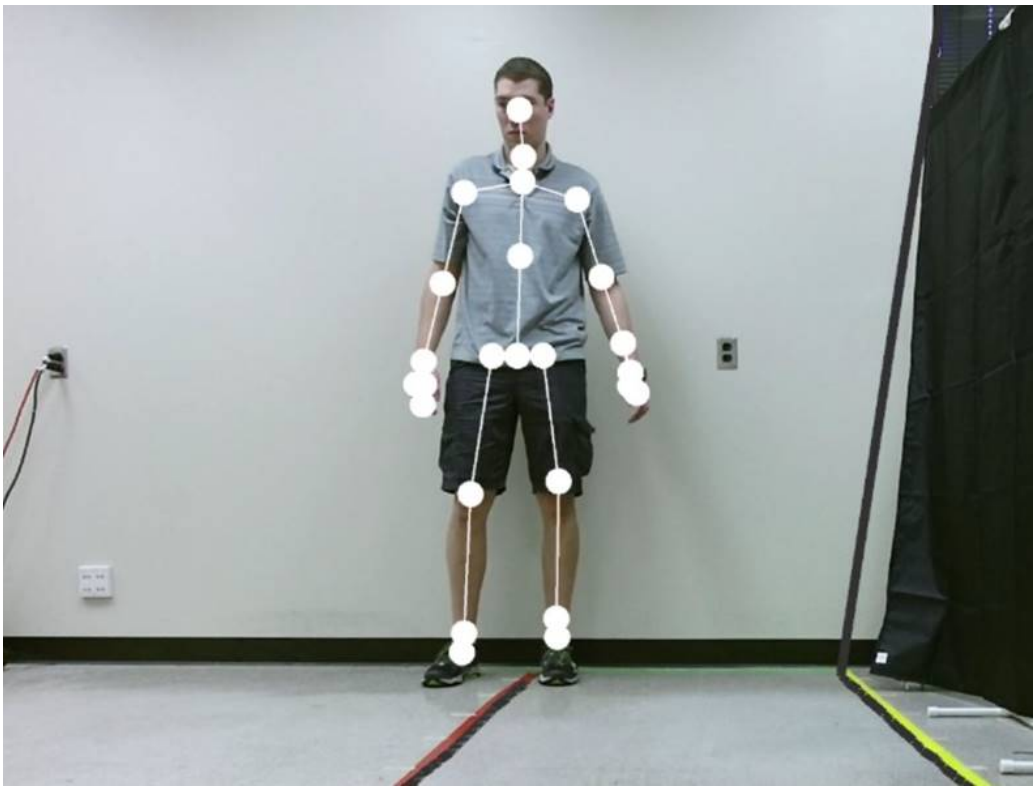

Figure 14: Kinect image showing body tracking using the Kinect PV2 library

Algorithm 8 provides a high level description of how this library works.

Algorithm 8 Body tracking using the KinectPV2 library

```
1:  import KinectPV2.KJOINT; //Initialize body tracking libraries
2:  import KinectPV2.*;
3:  kinect = new KinectPV2(this); // initialize a Kinect object
4:  kinect.enableSkeletonColorMap(true); // enable skeleton tracking
5:  kinect.enableColorImg(true); // enable color image
6:  kinect.init(); // initialize the Kinect
7:  if skeleton.isTracked() // check to see if a body is being tracked then
8:      joints = skeleton.getJoints(); // if a body is being tracked, then the joint coordinates are stored
9:      color the joints a unique color.
10:     drawBody(joints) // this function draws lines of the of the chosen color between each of the joints of the body by
        accessing their X and Y coordinates
11: end if
```

The complex body-tracking algorithms developed by Microsoft Research enable the Kinect to detect skeletal joints from a single depth image. Shotton et. al. [14] demonstrated how they achieved highly accurate, whole-skeleton tracking.

4.2 Pedestrian-Avoidance

Once successful lane keeping was achieved with the test vehicle, testing began on the pedestrian-avoidance capabilities.

Algorithm 9 Pedestrian avoidance

```
1: Perform standard lane keeping algorithm (Algorithm 3)
2: if Pedestrian Detected then
3:     Calculate the X and Y coordinates of pedestrian's left and right foot
4:     if leftFootX-roadCenter < roadWidth/2 || rightFootX-roadCenter < roadWidth/2 then
5:         directionVariable = 3; // The switch statment that uses directionVariable was explained in Algorithm 7 and stops
           the vehicle if the value is 3
6:     else
7:         Continue forward
8:     end if
9: else
10:    Follow the path as described in the previous algorithm until a pedestrian is detected
11: end if
```

Algorithm 9 combined the previous lane keeping algorithm and body-tracking algorithm to create a pedestrian-avoidance algorithm. This algorithm checked the location of a pedestrian's left and right feet to see if the person was located at an offset from the center of the road that was less than half the width of the road, and if so, stopped the test vehicle until the pedestrian exited the path. Algorithm 9 was shown to consistently avoid collisions with pedestrians during testing.

5 EXPERIMENTAL RESULTS

This thesis successfully demonstrated path navigation, lane-keeping, and pedestrian-avoidance using the visual data captured from the Kinect. This section provides an analysis of the results of the lane keeping and pedestrian avoidance experiments. These experiments were conducted using the test track diagrammed in figure 15. The test track was configured to test lane-keeping for a range of road curvatures. The 8 m track began with a short straightaway (20% of path), transitioned into a gradual curve (20% of path), followed by a sharp curve (20% of path), then finished with a longer straightaway (40% of path).

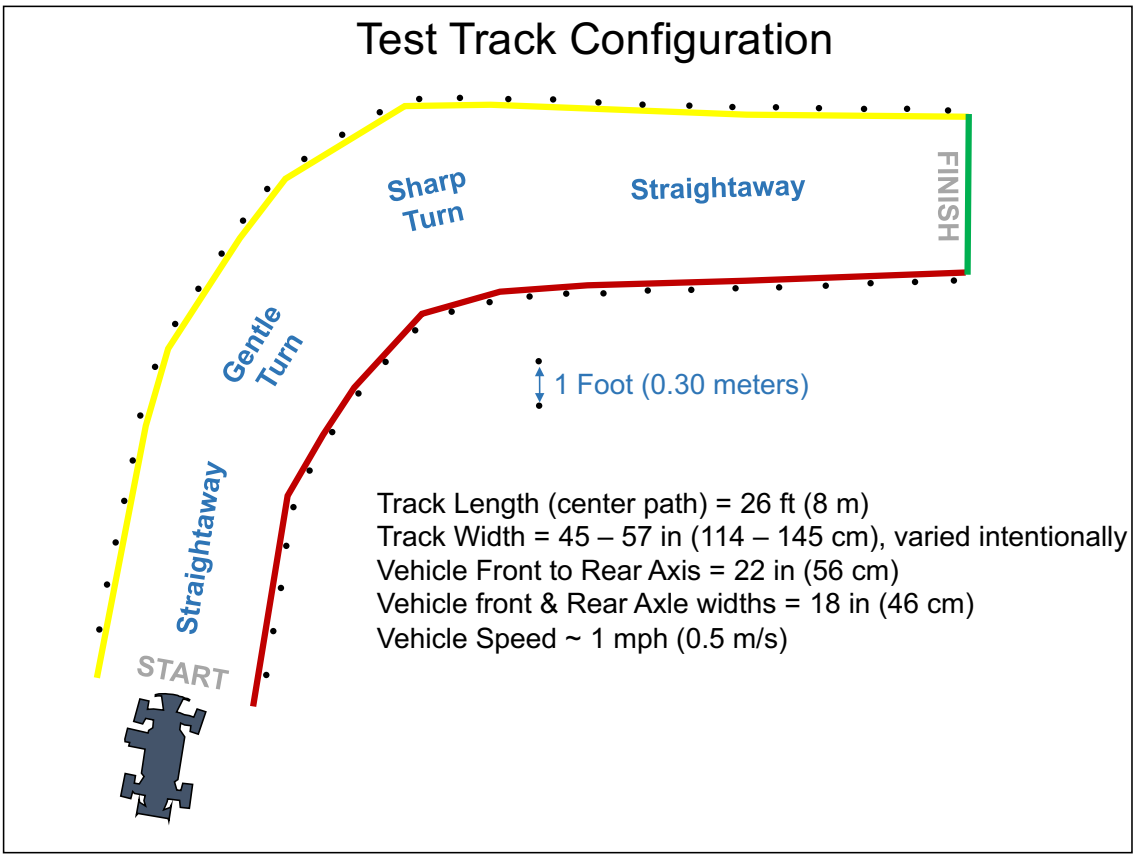

Figure 15: Diagram of the track used during testing

5.1 Lane-Keeping

Lane keeping was performed using a PD controller to generate a steering angle for the test vehicle as discussed in Sections 3 and 4. The following section demonstrates the empirical process used to determine the optimal controller gains.

5.1.1 PD Controller Gain Tuning

The gains within the steering angle controller had to be tuned in order to optimize the performance of the test vehicle. This was accomplished by gathering data for a range of gains and then choosing the gains that minimized operational error. The P gain, K_p, was tested first by setting the D gain, K_d, to zero. Once an optimal P gain was found, then a range of D gains was tested at optimal P gain to determine the combination of P and D gains that would minimize error.

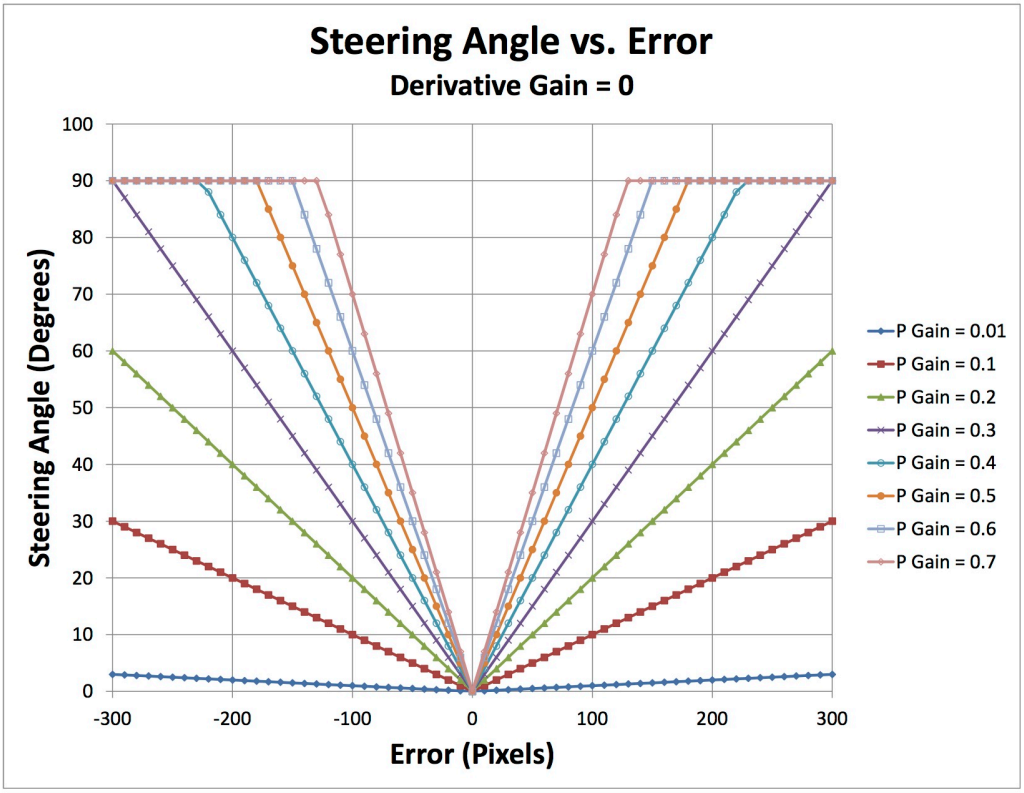

Figure 16: Steering angle vs. position error for multiple P gains

Figure 16 is a simulation of the absolute value of the steering angle deviation from the neutral position as error changes for a range of P gains at a D gain of zero. Since a proportional controller is basically the equation of a line with the gain acting as the slope, it was simple to plot a simulation of steering angle values for a range of errors. This plot was useful for determining the range of P gains that would result in a set of steering angles falling within the operating range of the test vehicle. Based on the simulation, P gains varying from 0.1 to 0.7 were chosen for actual testing.

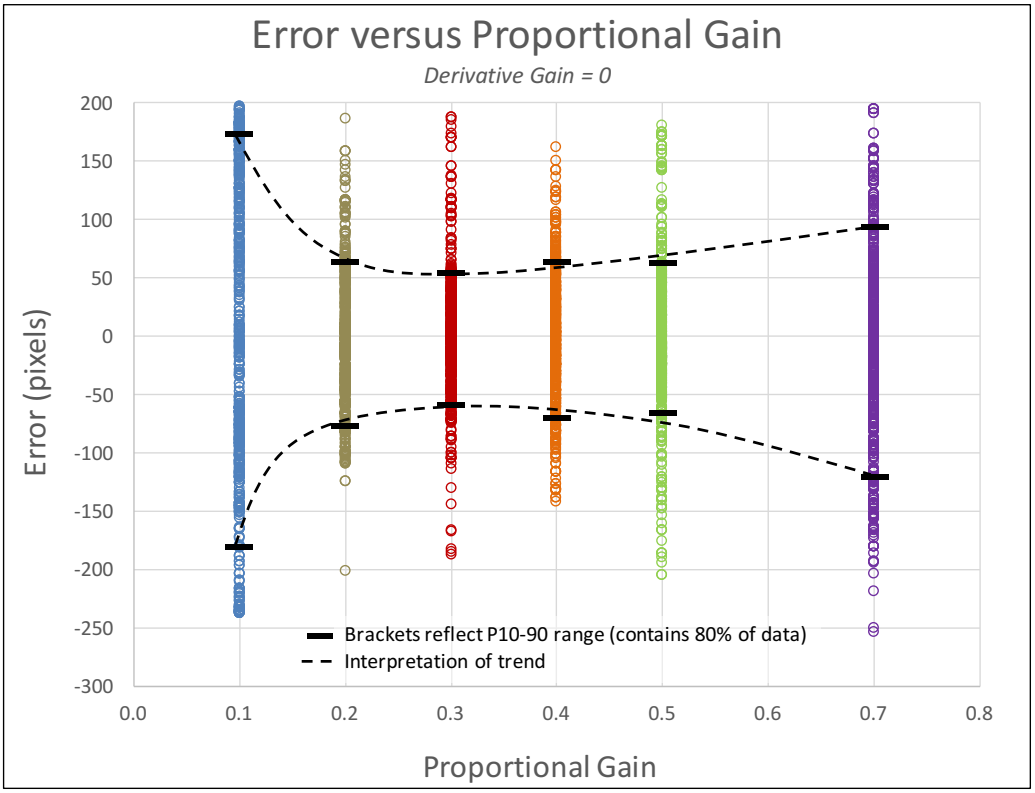

Figure 17: Experimental data showing vehicle position error vs. proportional gain with zero D gain

Figure 17 was generated using the data from 18 test runs, three for each for seven P gain values. The plot displays the range of vehicle position error generated along the entire test track. To improve statistical reliability, this plot contains data from three individual test runs for each P value. The data within the black bars on each P gain represents the 10% to 90% probability range (P10-P90) of the data points gathered during testing. 10% of the data points on each end of the error spectrum were removed to eliminate outliers from the analysis, and error ranges of the central 80% of the data were compared. Potential causes for these outliers include variable lighting conditions and vibration of the Kinect, both of which will be discussed toward the end of this section. The P gain value that displayed the

lowest overall error within the P10-P90 range was 0.3.

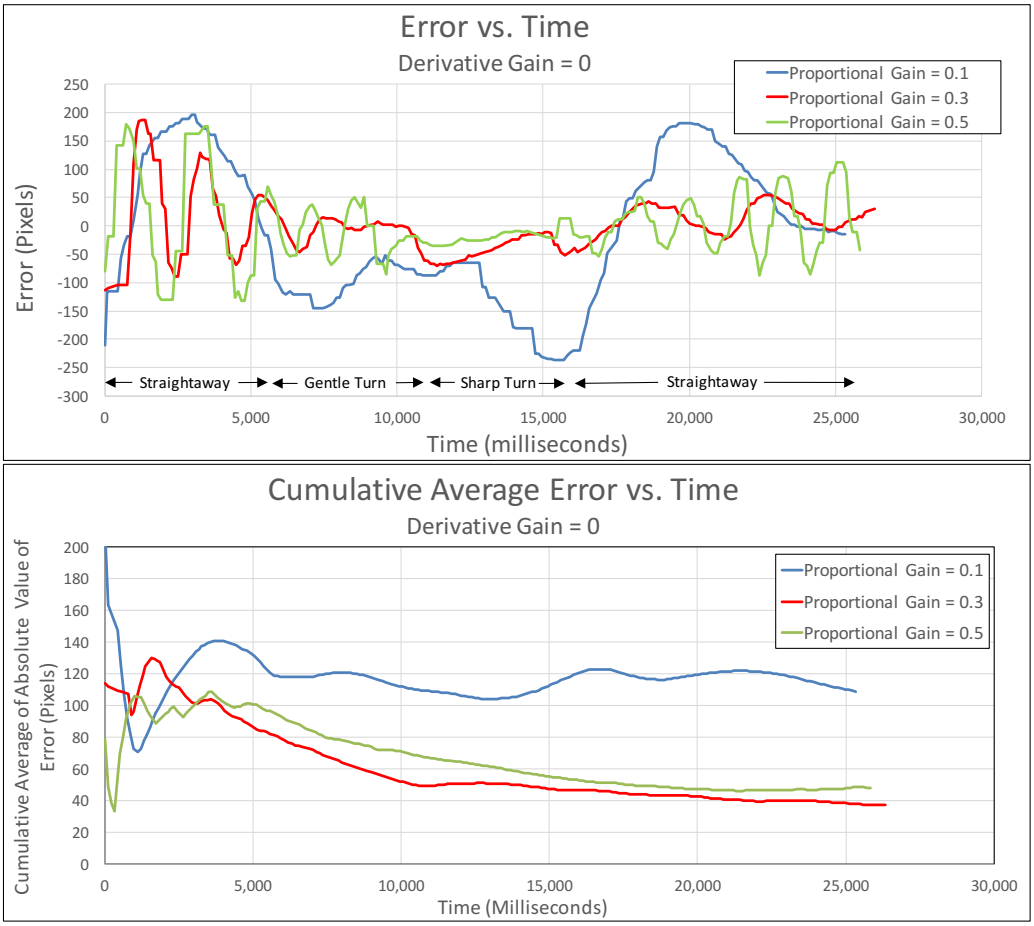

Figure 18: Test runs comparing error vs. time and cumulative average error vs. time for varying P gains with zero D gain

Figure 18 displays the results of three test runs for P gains of 0.1, 0.3, 0.5. These plots demonstrate how a P gain of 0.3 improves the error performance compared to P gains of 0.1 and 0.5. It is clear that the P gain of 0.1 was too low, and resulted in low frequency high magnitude oscillations while the P gain of 0.5 was too high and resulted in unstable, high frequency oscillations around the road center. The cumulative average error vs time plot compares the same three test runs and

shows that the average error for P = 0.3 is lower than P = 0.1 and P = 0.5.

As mentioned previously, the P gain of 0.1 caused the test vehicle to perform poorly for all segments of the path, and the test vehicle was almost unable to make it to the end of the track without leaving the path boundaries. The test for the P gain of 0.5 struggled the most with the straightaways. At the beginning of the test the vehicle experienced high frequency oscillations, during the curved section in the middle of the test the P gain of 0.5 performed almost as well as the optimal gain of 0.3, but then returned to unstable oscillations of increasing magnitude after transitioning back to the final straightway. From the data it is clear that a P gain of 0.3 resulted in the most consistently low error over the entire path. After choosing 0.3 as the P gain, it was then necessary to vary the D gain until the system was critically damped.

After adding the derivative gain to the controller, generating a simulated plot similar to Figure 16 would have required a complex simulator to be built. This is because the D term provided the controller with damping. So instead of using a simulation, Figure 19 was generated using the data from actual experiments.

Adding the D term causes the controller to return steering angles that are no longer directly proportional to the error, forming a cloud of points instead of a straight line.

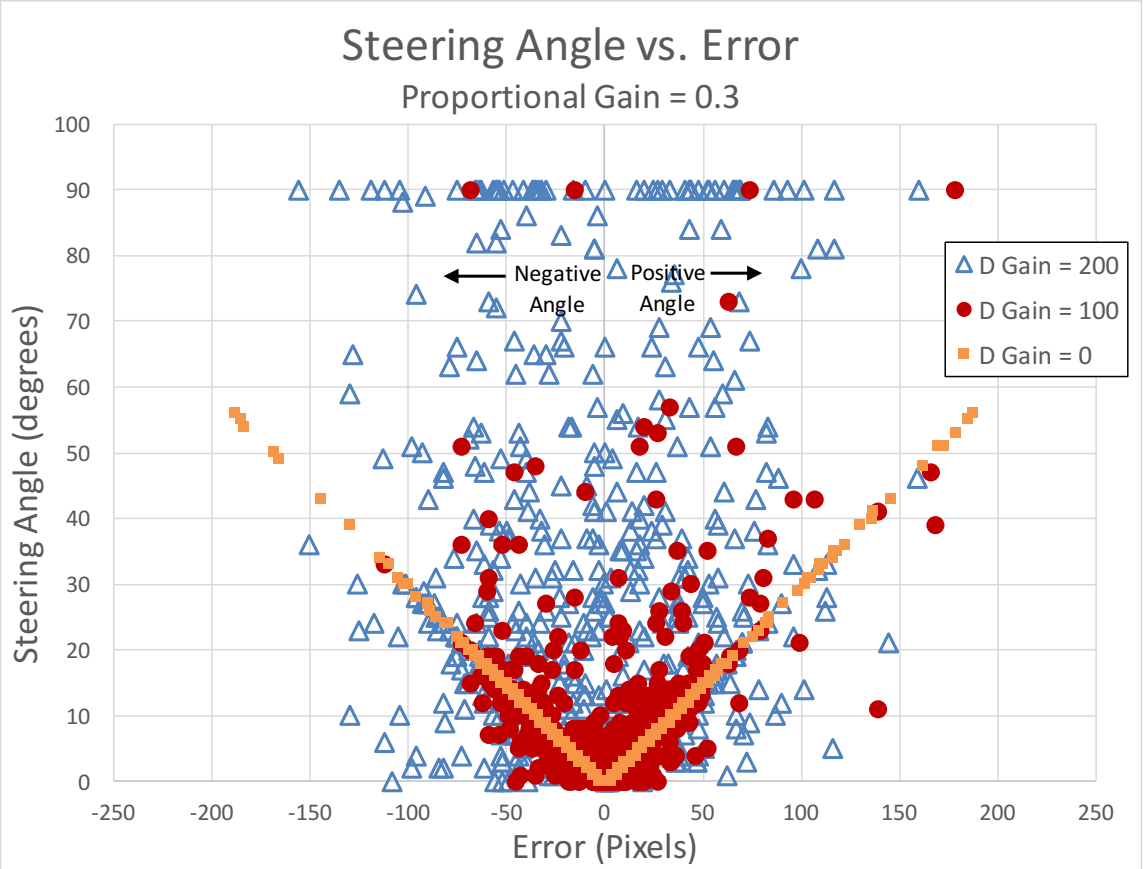

Figure 19: Experimental data showing steering angle vs. D gain for P gain of 0.3

Lacking a simulation to determine a viable range of D values to test, a broad range of D gains were chosen. Figure 20 shows a comparison of magnitude of the vehicle position error over the entire test track as the D gain is varied while holding the P gain constant at the optimal value of 0.3.

Fourty-two test runs were made to evaluate D gain values ranging from 0 to 500. The vehicle position error for 7 of those D gain values are shown in Figure 20.

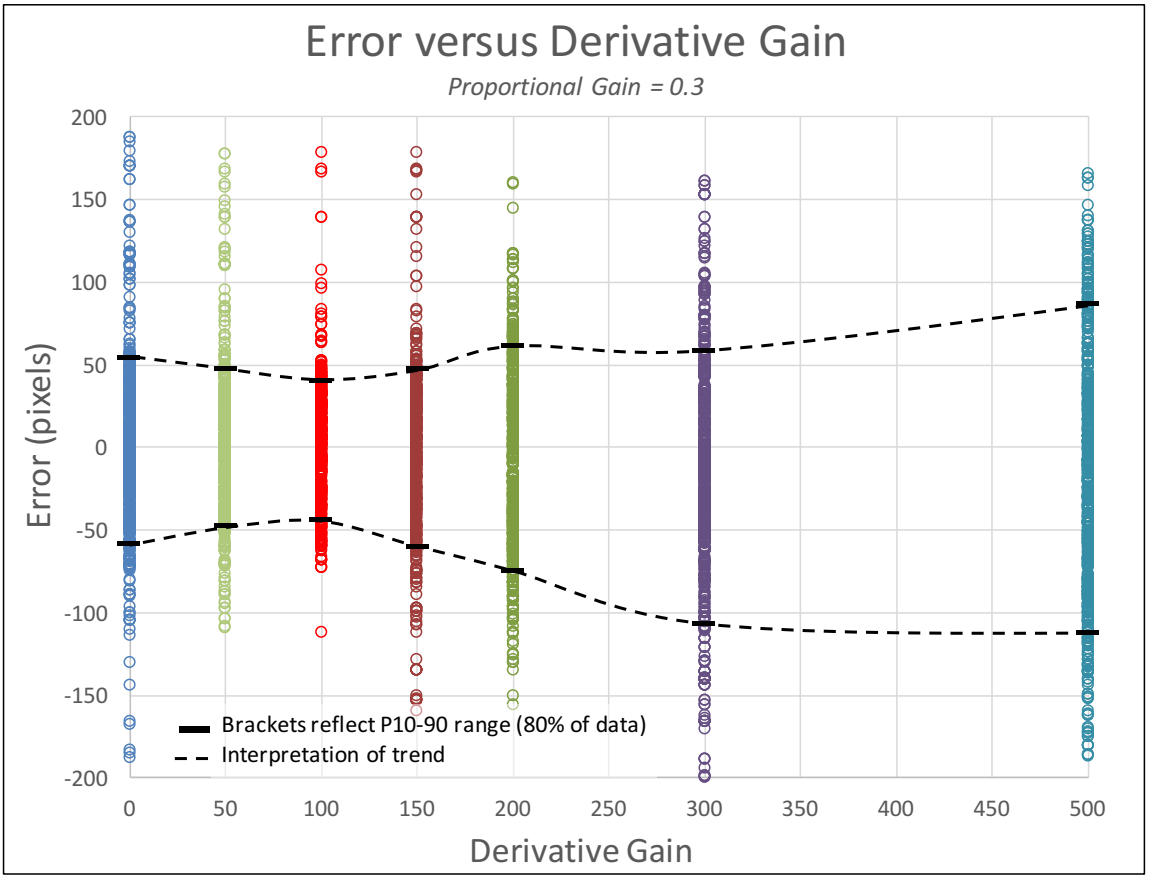

Figure 20: Experimental data showing vehicle position error vs. derivative gain with P gain of 0.3

Data from three individual test runs for each D gain were included in the plot in order to improve statistical reliability. As with Figure 17, the optimal D gain was selected based on the P10-P90 probability range of the data in order to eliminate the influence of outliers. The data indicated that a D gain of 100 represents the most consistently low vehicle position error.

The plots in Figure 21 display the results of three test runs for D gain values of 0, 100 and 300. These plots demonstrate how a D gain of 100 improves the error performance compared to D gains of 0 and 300.

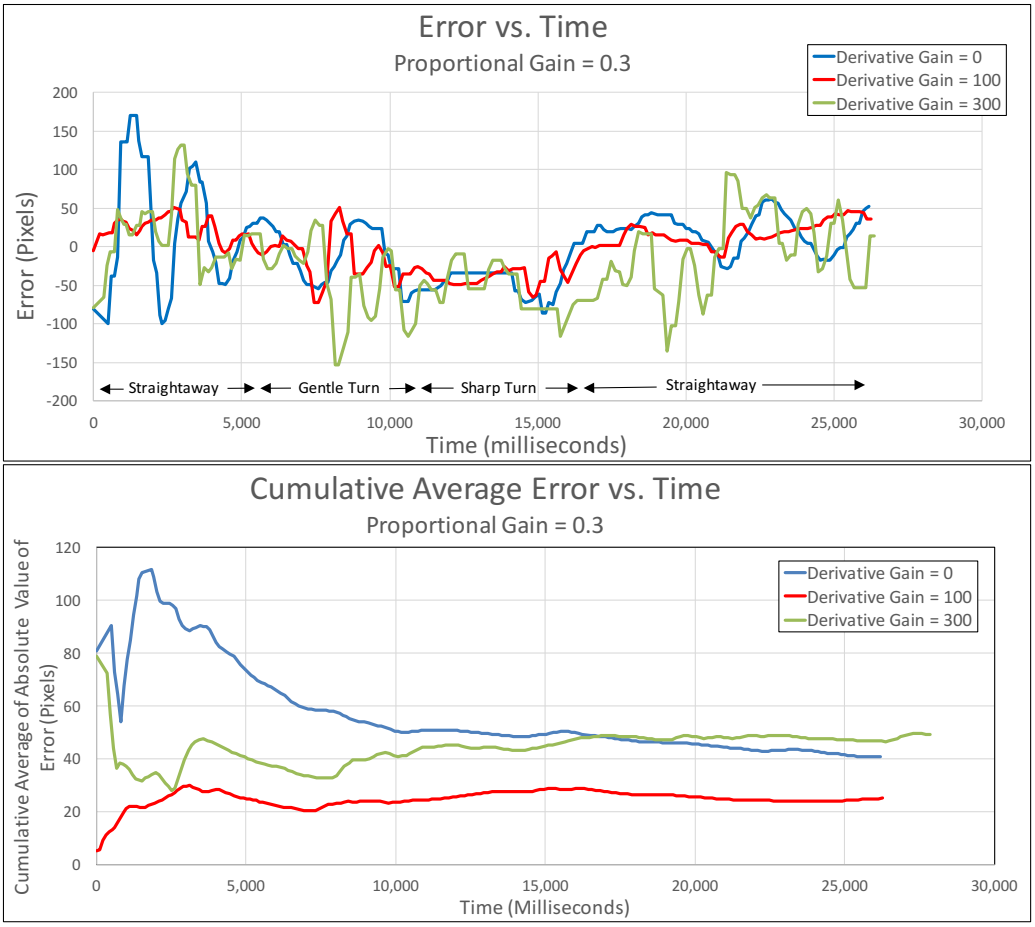

Figure 21: Test runs comparing error vs. time and cumulative average error vs. time for varying D gains with P gain = 0.3

Increasing the D gain to 300 resulted in a large number of sharp, imprecise movements of the vehicle that generated a significant position error. The error vs. time plot for a D gain of 300 shows these sharp movements in the form of high frequency oscillations throughout the entire test.

For all of the gains tested, it was noted the error was consistently at its lowest during the sharp turn section of the track. This is because the steering angle gen-

erated by the controller was never high enough for the front center point of the vehicle to cross over the center of the road. This means that error is always negative during the sharp turn, so there are no oscillations around the center of the road which resulted in low error for that section of the test track.

Out of all of the values tested, a D gain of 100 resulted in the lowest average error. As a result of the testing, the gains for the PD controller were chosen to be a P gain of 0.3 and a D gain of 100. Lane keeping test with the optimal gains were able to achieve an error of less than 50 pixels or approximately 3.6 cm for the majority of the test track.

5.2 Pedestrian-Avoidance

The pedestrian avoidance algorithm was successfully able to detect and stop to avoid a pedestrian that was blocking the path during testing. The results of this algorithm can be most succinctly demonstrated by viewing the video referenced in [33]. The response time of the pedestrian avoidance algorithm relies entirely on the Kinect's ability to successfully detect a human body. This response time was tested by repeatedly walking in front of the Kinect and timing how long it took from the moment the pedestrian entered the field of view until the tracking algorithm detected him. Figure 22 shows the detection time for 10 test runs for each of two scenarios. In the first scenario, the Kinect and the test vehicle were motionless while a pedestrian walked into the view range. During the second test, the test vehicle was moving while the pedestrian walked in front of the vehicle.

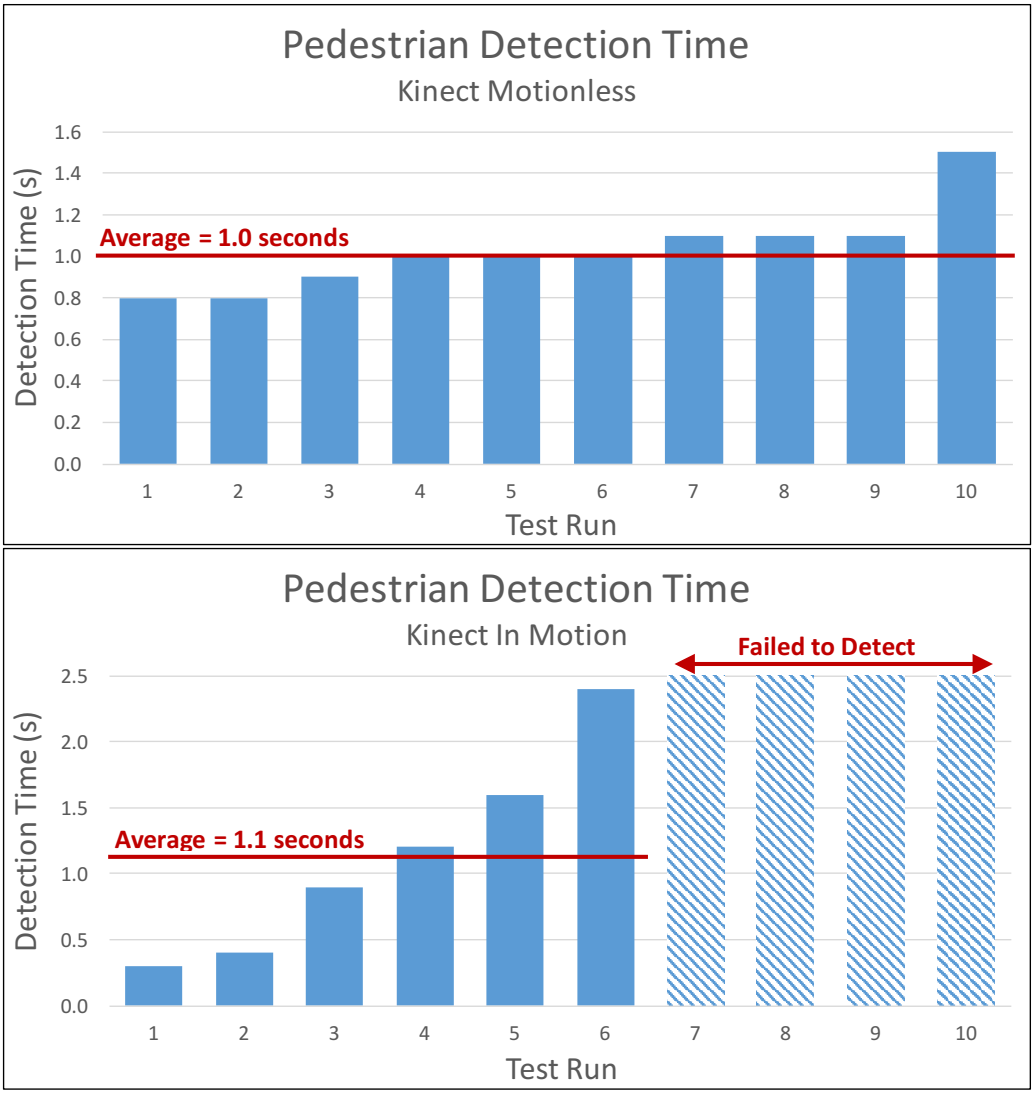

Figure 22: Average time for the Kinect to detect a pedestrian while the Kinect is motionless and in motion

The data show that the Kinect is far less reliable at detecting pedestrians when it is moving. The Kinect was designed to track human motion to be used for control of video games, and therefore the Kinect is intended to be motionless and placed near a television during operation [14]. When the Kinect is in motion, the pattern recognition algorithms have trouble properly identifying and tracking a human

body. For 10 experiments run while the Kinect was in motion, a 40% failure rate for pedestrian detection was observed, and while motionless, the failure rate was 0%. For the runs that detected a pedestrian successfully, the average detection times for each case were similar at 1.0 s when the Kinect was motionless and 1.1 s while the Kinect was in motion. However, the data gathered while the Kinect was moving was very inconsistent. When motionless, the pedestrian detection time ranged from 0.8 s to 1.5 s, and while the Kinect was moving, the detection time varied from 0.3 s to 2.4 s. A potential reason for some of the surprisingly low detection times seen only when the Kinect was moving is that the view range was also moving, thereby occasionally orienting the detection camera more centrally on the pedestrian as he approached the lane, thereby providing a biased advantage to the detection time.

If the detection reliability could be improved while the Kinect was in motion, the above detection times would be adequate to avoid collisions at the low speeds at which the tests were conducted (approximately 0.5 m/s). However, if the test vehicle was operating at higher speeds, the range of detection delays discussed above could result in a collision with the pedestrian which would be unacceptable. Increasing the speed of the machine learning algorithms used for human tracking was beyond the scope of this thesis.

An ideal test run would have the Kinect detect the pedestrian as soon as he entered the detection range of the camera, then continued to track his body until he leaves the detection range completely. All of the tests conducted while the Kinect was not moving followed this pattern. However, for the tests conducted while the Kinect and test vehicle were in motion, 90% of the tests showed the Kinect losing track of the pedestrian before he exited the detection range.

5.3 Effects of Hardware and Software Limitations on Data

This section discusses the Kinect's ability to reliably and consistently gather data. During testing, the Kinect's limited field of view, varying light conditions, and movement of the test vehicle interfered with data collection and resulted in many data sets being discarded early in the project. The following sections discuss these limitations.

5.3.1 Kinect's Limited Field of View

While gathering data, it became clear that certain hardware and software limitations of the Kinect and test vehicle affected the accuracy of the data gathering process. In Section 2, it was mentioned that the Kinect has a horizontal view range of 70 degrees and a vertical view range of 60 degrees. For lane keeping and pedestrian avoidance data to be gathered accurately, it was necessary for both boundaries of the path to be within view of the Kinect's cameras at all times, and for the Kinect to be able to identify a human standing in the path. For these conditions to be met, the Kinect had to be elevated off of the test vehicle, and tilted up at a small angle so that it was possible to view the full body of a pedestrian while still being able to see the boundaries of the path on the ground. Tilting the Kinect up adjusted the field of view to better detect a human, but at the same time it limited the range of view of the path boundaries and changed the distance at which the Kinect was able to detect the path boundaries. This is relevant because when performing lane keeping, the Kinect checks a point as close to the center of the front bumper of the vehicle as possible in order to minimize position error. Increasing the distance of this point from the vehicle increases the chance that the vehicle may leave the boundaries of the path while turning, even though the tracking algorithms show low position error.

5.3.2 Impact of Varying Light Conditions

Another important consideration was the effect that the lighting conditions had on the Kinect's ability to identify the pixels that made up the boundary of the path that the vehicle followed. Misidentification of the road boundary pixels was the most frequent cause of experiment failure early in the project. If the Kinect was suddenly unable to properly identify one or both of the boundaries of the path, then the calculation for the location of the center of the path would be incorrect. The test vehicle would then use that incorrect value as the reference point for the controller which often resulted in the vehicle driving outside of the path. Figure 23 shows an example of this phenomena.

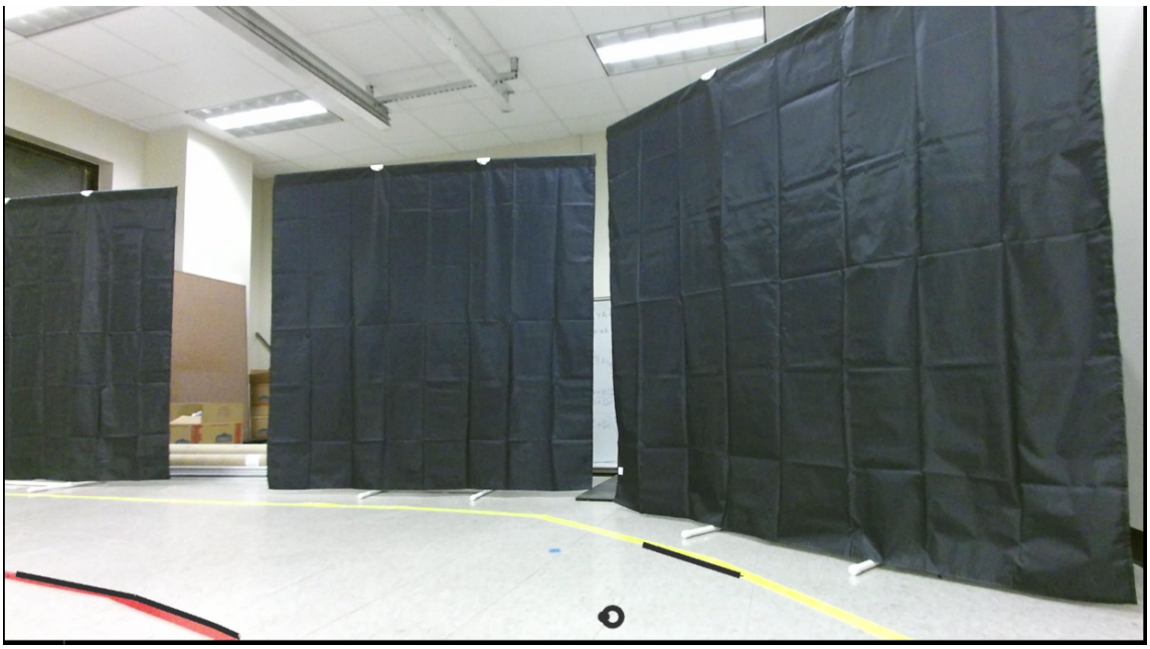

Figure 23: Example of a path detection failure

The combination of bright light and a high angle that the Kinect was pointed sometimes resulted in the Kinect losing track of the path boundaries. Shortly after this photograph was taken, the test vehicle continued traveling forward and collided with the curtain outside of the path.

5.3.3 Effects of Motion on Body Tracking

The Kinect was able to identify and track a human body within its field of view with high reliability as long as it could view the entire body from head to feet, and the Kinect was motionless. During testing, the negative effects of moving the Kinect sensor became apparent. Movement of any kind appeared to confuse the machine-learning algorithms that the human tracking libraries are built on into thinking that random inanimate objects were actually humans. At various times, the Kinect incorrectly identified cardboard boxes, a metal cart, an aluminum structure, a dark curtain and a blank wall as humans and attempted to track them. This was problematic because if the Kinect misidentified a random object as a human, the tracking algorithms would often map the “joints” within the boundaries of the road and the test vehicle would stop to avoid a collision even though the road was clear. Also, as the vibrations from movement increased, the Kinect would frequently lose track of a human that it had already successfully identified. As shown in the second photo in Figure 24, the joint-tracking software would sometimes jump off of the human body and become stuck to a wall behind the person. Unfortunately, tuning the body detection software developed by Microsoft so that it is better able to detect bodies in motion is beyond the scope of this project.

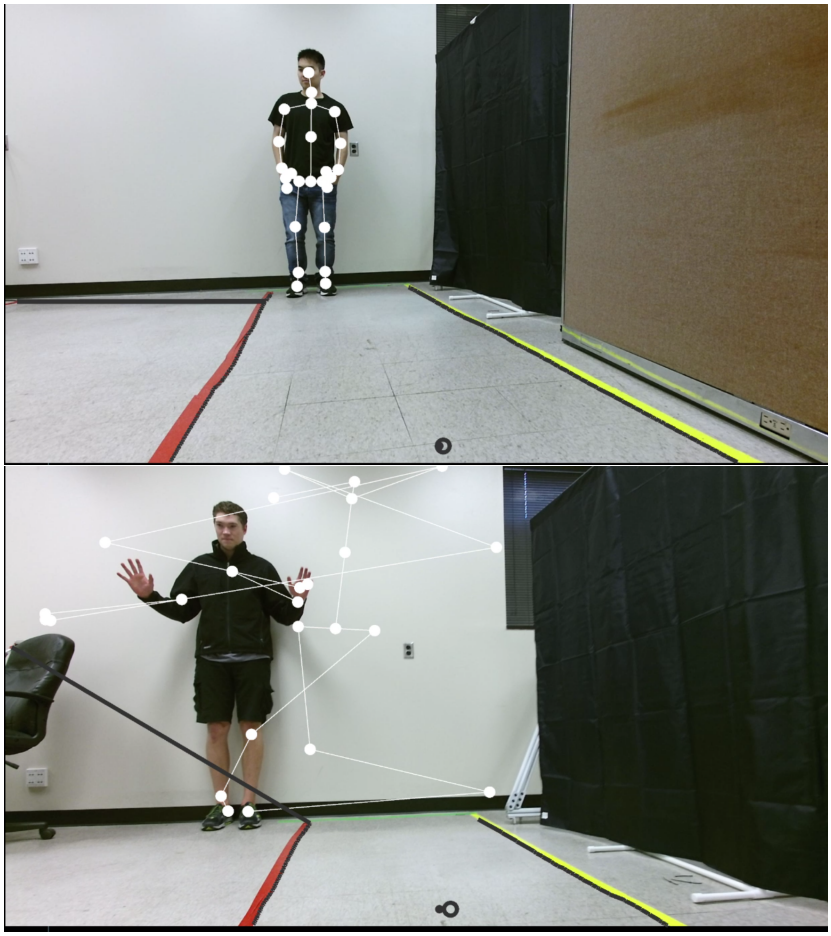

Figure 24: Examples of correct and incorrect body tracking using the Kinect PV2 library

The following section discusses conclusions drawn from the results shown in this section and presents options for continuation of this project in the future.

6 CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

The test vehicle developed for this thesis successfully performed lane keeping at low speeds with a steady-state error of less than 75 pixels (5cm) after the PD controller for the steering servo was properly tuned. The test vehicle demonstrated performance that was reliable enough to be used for simple applications like product delivery in warehouses, using dedicated robot pathways where there is no need to worry about pedestrian detection.

The success of the pedestrian avoidance algorithm used in this project was found to be limited by the constraints of the Kinect's hardware and machine learning algorithms. For the pedestrian avoidance algorithm developed for this thesis to be applicable in real world situations, the reliability of the Kinect's human detection capabilities would have to be improved for situations where the Kinect is in motion. If the Kinect were able to detect and track human bodies with the same 100% accuracy observed while the Kinect was non moving instead of the current 40% failure rate while in motion, then the pedestrian avoidance algorithms developed for this thesis could be used to increase the safety of product transport vehicles in warehouses, hospitals, and other facilities where mobile robots operate in proximity to humans. The principles for the pedestrian-avoidance algorithm developed in this thesis could also potentially be used for autonomous pedestrian avoidance in automobiles if the detection of distance, loop rate, and body tracking reliability were improved.

In summary, the hardware used for this project was not robust enough for reliable human tracking, but the algorithms for lane keeping and human tracking were

demonstrated to be successful.

6.2 Future Work

There are several improvements that could be made to the test vehicle that would increase its effectiveness in both lane keeping and pedestrian avoidance.

6.2.1 Hardware Improvements

As was mentioned in Section 4, it was difficult to adjust the Kinect so that it was effectively able to track and avoid pedestrians while navigating the path due to the Kinect's limited field of view. A potential solution to this problem would be to use two separate cameras, one for lane keeping and another one for pedestrian avoidance. That way, each camera could be pointed at tilt angles to allow the lane keeping and pedestrian avoidance algorithms to perform optimally.

Another hardware improvement would be to embed a computing system and a battery for the Kinect into the test vehicle. Doing so would allow Processing functions to run on-board, and would eliminate a large number of cables dragging behind the vehicle during testing.

6.2.2 Software Improvements

Adding an obstacle avoidance algorithm would increase the number of applications that this system could be used for. Addition of this algorithm would enable the Kinect to navigate more complex paths and react to common obstacles encountered in warehouses, hospitals and other environments.

It would also be beneficial to add a velocity controller. This controller would require real-time feedback that could be achieved through a high resolution encoder. A velocity controller would allow the Kinect to optimize its speed during test runs by

increasing vehicle velocity when the road is clear and then slowing the vehicle to improve reaction time if it detected pedestrians or potential obstacles. Successful implementation of a velocity controller would require the PD controller on steering to be re-tuned for variable speeds.

Another improvement would be to enable the test vehicle to travel on bumpy roads or roads with variable curvature. This could be done by using the Kinect's depth map in conjunction with the color images to create a better understanding of the terrain in front of the vehicle.

REFERENCES

[1] Kawazoe, H., Murakami, T., Sadano, O., Suda, K. et al., “Development of a Lane-Keeping Support System,” SAE Technical Paper 2001-01-0797, 2001, doi:10.4271/2001-01-0797.

[2] K. Naab and G. Reichart, “Driver Assistance Systems for Lateral and Longitudinal Vehicle Guidance,” in International Symposium on Advanced Vehicle Control, 1994, doi:00786847

[3] S. J. Anderson, S. C. Peters, T. E. Pilutti, K Iagnemma, “An optimal-control-based framework for trajectory planning, threat assessment, and semi-autonomous control of passenger vehicles in hazard avoidance scenarios,” in International Journal of Vehicle Autonomous Systems, 2010, doi:10.1504/IJVAS.2010.035796

[4] J. Chiu, S. Solmaz, M. Corless, and R. Shorten, “A methodology for the design of robust rollover prevention controllers for automotive vehicles using differential braking,” in International Journal of Vehicle Autonomous Systems, 2010, doi:10.1504/IJVAS.2010.035794

[5] GM Authority Staff. (2014). *GM’s Lane Departure Warning and Lane Keep Assist Tech: Feature Spotlight* [Online]. Available: http://gmauthority.com/blog/2014/11/gms-lane-departure-warning-and-lane-keep-assist-tech-feature-spotlight/ [Accessed: 05- Jun-2016].

[6] Ford Motor Company. (2016) *Lane Keeping System* [Online]. Available: https://owner.ford.com/how-tos/vehicle-features/safety/lane-keeping-system.html. [Accessed: 05-Jun-2016].

[7] Toyota. (2016). *Lane Keeping Assist* [Online]. Available: http://www.toyota-global.com/innovation/safety_technology/safety_technology/technology_file/active/lka.html. [Accessed: 05-Jun-2016].

[8] Lincoln Motor Company. (2016). *Lane-Keeping System* [Online]. Available: http://www.lincoln.com/cars/mks/features/?featID=3#page=Feature21. [Accessed: 05-Jun-2016].

[9] Google. (2016). *Google Self-Driving Car Project* [Online]. Available: https://www.google.com/ selfdrivingcar/how/ [Accessed: 10-Mar-2016].

[10] D. Sherman. (2015). *Elon, Take the Wheel! We Test Tesla's New Autopilot Feature* [Online]. Available: http://blog.caranddriver.com/elon-take-the-wheel-we-test-teslas-new-autopilot-feature/. [Accessed: 05-Jun-2016].

[11] Microsoft. (2016). *Kinect hardware* [Online]. Available: https://developer.microsoft.com/en-us/windows/kinect/hardware [Accessed: 05-Jun-2016].

[12] Asus. (2016). *Xtion PRO LIVE - Overview* [Online]. Available: https://www.asus.com/us/3D-Sensor/Xtion_PRO_LIVE/ [Accessed: 01-Jun-2016].

[13] R. Cong and R. Winters. (2016). *How It Works: Xbox Kinect* [Online]. Available: http://www.jameco.com/jameco/workshop/howitworks/xboxkinect.html [Accessed: 05-Jun-2016].

[14] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake, "Real-Time Human Pose Recognition in Parts from Single Depth Images",*Computer Vision and Pattern Recognition*., Colorado Springs, CO, 2011, pp. 1-3.

[15] Y. Li, “Hand Gesture Recognition Using Kinect,” M.S. thesis, Dept. Comp. Eng. and Comp. Sci., Univ. of Louisville, Louisville, KY, 2012

[16] Chi-Ying Liang and Huei Peng, ”Optimal Adaptive Cruise Control with Guaranteed String Stability” in Vehicle System Dynamics, Vol. 32 no.4–5, pp. 313–330.

[17] Chi-Ying Liang and Huei Peng, ”String Stability Analysis of Adaptive Cruise Controlled Vehicles,” *JSME International Journal Series*, vol. 43, no. 3, pp. 671–677.

[18] NHTSA. (2015, February). *Critical Reasons for Crashes Investigated in the National Motor Vehicle Crash Causation Survey* [Online]. Available: http://www-nrd.nhtsa.dot.gov/pubs/812115.pdf. [Accessed: 05-Jun-2016].

[19] G. Trowbridge. (2016). *U.S. DOT and IIHS announce historic commitment of 20 automakers to make automatic emergency braking standard on new vehicles | National Highway Traffic Safety Administration (NHTSA)* [Online]. Available: http://www.nhtsa.gov/About+NHTSA/Press+Releases/nhtsa-iihs-commitment-on-aeb-03172016 [Accessed: 05-Jun-2016].

[20] Ford. (2016). *Ford Tripling Autonomous Vehicle Development Fleet, Accelerating On-Road Testing of Sensors and Software* [Online]. Available: https://media.ford.com/content/fordmedia/fna/us/en/news/2016/01/05/ford-tripling-autonomous-vehicle-development-fleet–accelerating.html [Accessed: 10-Mar-2016].

[21] A. Davies. (2015, October 15). *GM Has ‘Aggressive’ Plans for Self-Driving Cars* [Online]. Available at:

http://www.wired.com/2015/10/gm-has-aggressive-plans-for-self-driving-cars/ [Accessed: 10-Mar-2016].

[22] Nissan USA. (2014, March 15). *Nissan's Self Driving Car* [Online]. Available at: http://www.nissanusa.com/blog/autonomous-drive-car [Accessed: 10-Mar-2016].

[23] D. Tam. (2014). *Meet Amazon's Busiest Employee–the Kiva Robot* [Online]. Available: http://www.cnet.com/news/meet-amazons-busiest-employee-the-kiva-robot/. [Accessed: 05- Jun- 2016].

[24] Aethon. (2016). *TUG Robots in Healthcare* [Online]. Available: http://www.aethon.com/tug/tughealthcare/ [Accessed: 05-Jun-2016].

[25] R. Hutchinson. (2013). *Xbox One Kinect Teardown (Video)* [Online]. Available: http://www.geeky-gadgets.com/xbox-one-kinect-teardown-video-22-11-2013/ [Accessed: 06-Jun-2016].

[26] M. Szymczyk. (2014). *How Does The Kinect 2 Compare To The Kinect 1?* [Online]. Available: http://zugara.com/how-does-the-kinect-2-compare-to-the-kinect-1 [Accessed: 05-Jun-2016].

[27] Redcat Racing. (2016). *Brushless inrunner motor B4485 for RC cars* [Online]. Available: http://www.redcatracing.com/manuals/07788manual.pdf [Accessed: 05-Jun-2016].

[28] Redcat Racing. (2016). *Rampage Chimera EP Pro 1/5 Scale Electric Sand Rail* [Online]. Available: http://www.redcatracing.com/Rampage-Chimera-Pro-Electric [Accessed: 05-Jun-2016].

[29] *Arduino Mega 2560* [Online]. Available at: https://www.arduino.cc/en/main/arduinoboardmega2560. [Accessed: Jan-2016].

[30] T. Lengeling. (2014). *Kinect v2 Processing library for Windows – Codigo Generativo* [Online]. Available: http://codigogenerativo.com/kinectpv2/ [Accessed: 10-Mar-2016].

[31] G. Franklin et al., “chapter,” in *Feedback Control of Dynamic Systems*, 5th ed. Upper Saddle River, 2006, ch. 4, sec. 3, pp. 186-191.

[32] N. Soufi et al., “A Parameter Varying PD Control for Fuzzy Servo Mechanism,” in Intelligent Control and Automation, 2014, doi:http://dx.doi.org/10.4236/ica.2014.53018

[33] Forrest Berg. (2016, Jun. 9). Lane Keeping and Pedestrian Avoidance for a Vision Based Autonomous Test Vehicle. [Youtube Video]. Available: https://www.youtube.com/watch?v=oGbs2tDV9QA. Accessed Jun. 9, 2016.

APPENDIX

A.1 Processing Code

```
/* Pedestrain Avoidance and Lane keeping Algorithms written by
Forrest Berg


 KinectPV2, Kinect for Windows v2 library for processing written by
 Thomas Sanchez Lengeling.
 http://codigogenerativo.com/

 */

import KinectPV2.KJoint;
import processing.serial.*;
import KinectPV2.KJoint;
import KinectPV2.*;

KinectPV2 kinect;

Serial myPort;



// Center of vehicle in camera view
int xCenter = 1920/2 + 50;
int yCenter = 1080/2 + 500;

// Color Tracking Variables
int centerYellowBoundary = 0;
int centerRedBoundary = 0;
int roadWidth = 0;
float steeringAngle;
int stop = 0;
int xError = 0;
int prevXError = 0;
int rowIndex;
int columnIndex;
int found = 0;
int colorError;
float pix;
float modulus;
float rowcalc;
float remainder;
float rownum; // starts at 0
int prevCol = 0;
int prevRow = 0;
int directionVariable = 0;
int time=0;
int currentTime = 0;
int prevTime=0;
int startTime = 0;
int timeScalar = 5;
int pixCounter;
int leftFootBoundaryIndexYellow;
int rightFootBoundaryIndexYellow;
int leftFootBoundaryIndexRed;
int rightFootBoundaryIndexRed;
int groundTolerance = 50;
int arraysize = 1920*1080;
int mX;
int mY;
int[] errorArray = new int[1];
```

```
int index = 0;
int count1 = 0;
int count2 = 0;
int prevfound = 0;
int firstpix;


// Ground Tolerances
// Yellow
int YellowTolerance1 = 200;
int redTolerance1 = 165;
int blueTolerance1 = 165;
// Red
int YellowTolerance2 = 150;
int redTolerance2 = 180;
int blueTolerance2 = 150;

// skeleton tracking variables
KJoint[] joints;
int skeletonDetected;

// plotter variables
PrintWriter output;

void setup()
{
  startTime = millis();
  size(1920,1700,P3D);
  output = createWriter("data.txt");
  kinect = new KinectPV2(this);
  kinect.enableSkeletonColorMap(true);
  kinect.enableColorImg(true);
  kinect.init();

  //Serial Communication Setup
  String portName = Serial.list()[0]; //change the 0 to a 1 or 2 etc. to match your port
  myPort = new Serial(this, portName, 9600);
}

void draw()
{
  background(0);
  image(kinect.getColorImage(), 0, 0, width, 1080);

  ArrayList<KSkeleton> skeletonArray =  kinect.getSkeletonColorMap();
  //individual JOINTS
  for (int i = 0; i < skeletonArray.size(); i++)
  {
    KSkeleton skeleton = (KSkeleton) skeletonArray.get(i);
    if (skeleton.isTracked())
    {
       skeletonDetected = 1; //  addition to Thomas Lengling code
       joints = skeleton.getJoints();
       color col = color(255,255,255);
       fill(col);
       stroke(col);
       drawBody(joints);
    }
    else skeletonDetected = 0; // addition to Thomas Lengling code
  }
```

```
  laneKeeping();
  plotter();
}

// This function is designed to plot the value of error over time
void plotter()
{
   int rectHeight = 1700-1080;
   int xAxis = 1080+rectHeight/2;
   int xLocation = width;
   time = second();

   // Drawing Plot Axis
   stroke(0,102,153);
   strokeWeight(2);
   line(100,1080,100,height); // Y axis
   stroke(0,102,153);
   strokeWeight(2);
   line(100,xAxis,width,xAxis); // X axis
   textSize(32);
   text("Position Error vs Time", width/2-50, xAxis-250);
   textSize(20);
   text(  "0", 80, xAxis+5);
   textSize(20);
   text("100", 60, xAxis-100);
   stroke(0,102,153);
   strokeWeight(2);
   line(100,xAxis-100,width,xAxis-100);
   text("200", 60, xAxis-200);
   line(100,xAxis-200,width,xAxis-200);
   text("-100", 50, xAxis+100);
   line(100,xAxis+100,width,xAxis+100);
   text("-200", 50, xAxis+200);
   line(100,xAxis+200,width,xAxis+200);

   // Plotting Data
   for (int i = 0; i < width; i+=10*timeScalar)
   {
       line(100+ i,xAxis + 5,100 + i,xAxis - 5);
       if (i > 0) text(i/timeScalar,80+i,xAxis + 30);
   }
   if (errorArray.length > width - 100)
   {
       for (int i = errorArray.length-1; i > width - 100; i--)
       {
           if (xLocation > 100)
           {
               stroke(255,0,0);
               fill(255,0,0);
               point(xLocation,xAxis+errorArray[i]);
               xLocation --;
           }
       }
   }
   else
   {
     for (int i = 0; i < errorArray.length; i++)
     {
         stroke(0,0,255);
         fill(0,0,255);
```

```
          if (abs(errorArray[i]) < 310) ellipse(100 + timeScalar*i,xAxis+errorArray[i],10,10);
      }
    }
    // the timeScalar is the multiplier for spacing between pixels. makes it easier to understand data
    // when it is spread out more
    fill(255,255,255);
}

// This program was used for calibration of the color detection for tape boundaries
void testColor(color pix)
{
  if(abs(green(pix) - red(pix)) <= 50 & abs(red(pix) - blue(pix)) >= 60 & abs(green(pix)
  - blue(pix)) >= 60)
  {
    found = 1;
  }
  else
  {
    found = 0;
  }
  print("Red: "); print(red(pix));
  print(" Green: "); print(green(pix));
  print(" Blue: "); print(blue(pix));
  print(" Found: "); println(found);
}

void mouseClicked()
{
 mX = mouseX;
 mY = mouseY;
 output.flush(); // Writes the remaining data to the file
 output.close(); // Finishes the file
 exit(); // Stops the program
}

// This program performs color identification of the path boundary pixels and  stores their
//locations in arrays
void laneKeeping()
{
  currentTime = millis();
  int [] rowArrayYellow = new int[arraysize];
  int [] columnArrayYellow = new int[arraysize];
  int [] rowArrayRed = new int[arraysize];
  int [] columnArrayRed = new int[arraysize];
  for( int y = 0; y<1080; y++)
  {
   for(int x = 0; x<1920; x++)
   {
       color pix = get(x,y);
       color prevPix = get(x-1,y);
       color nextPix = get(x+1,y);
       if(abs(green(pix) - red(pix)) <= 40 & abs(red(pix) - blue(pix)) >= 70 & abs(green(pix)
       - blue(pix)) >= 70)
       {
           if(abs(green(prevPix) - red(prevPix)) <= groundTolerance & abs(green(prevPix)
           - blue(prevPix))
           <= groundTolerance & abs(blue(prevPix) - red(prevPix)) <= groundTolerance)
           {
               columnArrayYellow[count1] = x;
               rowArrayYellow[count1] = y;
```

```
                count1++;
            }
        }
        if(abs(red(pix) - green(pix)) >= 60 & abs(red(pix) - blue(pix)) >= 50 & abs(blue(pix)
        - green(pix)) <= 30)
        {
            if(abs(green(nextPix) - red(nextPix)) <= groundTolerance & abs(green(nextPix)
            - blue(nextPix))
            <= groundTolerance & abs(blue(nextPix) - red(nextPix)) <= groundTolerance)
            {
               columnArrayRed[count2] = x;
               rowArrayRed[count2] = y;
               count2++;
            }
        }
     }
    }

   // This section is designed to find the x coordinates of each boundary of the road that are
   // in the same row as the center dot. This should allow for the vehicle to navigate based on
   // the position of the center of the camera.

    for(int i = 0; i < count1; i++)
    {
      if(rowArrayYellow[i] == yCenter) centerYellowBoundary = columnArrayYellow[i];
    }
    for(int i = 0; i < count2; i++)
    {
      if(rowArrayRed[i] == yCenter) centerRedBoundary = columnArrayRed[i];
    }

// Position error calculation and storage

    roadWidth = centerYellowBoundary - centerRedBoundary;
    int roadCenter = roadWidth/2 + centerRedBoundary;
    xError = roadCenter - xCenter;
    errorArray[index] = xError;
    errorArray = expand(errorArray,errorArray.length + 1);
    index ++;

// This method guarentees that the vehicle will stop permanently after seeing green tape.
    color centerPix = get(xCenter,yCenter);
    if (green(centerPix) - blue(centerPix) >= 70 && green(centerPix) - red(centerPix) >= 100)
    {
       stop = 1;
    }
// Serial communication cant send negative integers, so directionVariable is
// used to tell the arduino if error is positive or negative

    if (xError < 0)  directionVariable = 1; //right
    else directionVariable = 2; // left

// Pedestrian detection and avoidance algorithm
    if (skeletonDetected == 1)
    {
       float leftFootX = joints[KinectPV2.JointType_FootLeft].getX();
       float rightFootX = joints[KinectPV2.JointType_FootRight].getX();

       if (abs(leftFootX-roadCenter) < roadWidth/2)
       {
```

```
        println("Pedestiran inside boundaires. Stopping Vehicle");
        directionVariable = 3;
     }
     else if (abs(rightFootX-roadCenter) < roadWidth/2)
     {
        println("Pedestrian inside boundaires. Stopping Vehicle");
        directionVariable = 3;
     }
     else if (stop == 0)
    {
       println("Road clear");
    }
  }

// These statements stop the car until it properly detects the road boundarires
  if (roadCenter < centerRedBoundary | roadCenter > centerYellowBoundary | abs(xError) > 800 )
  {
     directionVariable = 3; // stop indication set
     xError = 0;
  }

  if (stop == 1)
  {
     println("End of path recognized. Stopping Vehicle.");
     directionVariable = 3;
  }

// Sending data to the Arduino via serial port
  steeringAngle = .3*xError + 100*(xError - prevXError)/(currentTime-prevTime);
  myPort.write(abs(xError)); // negative numbers dont work. going to need to send a direction variable
  myPort.write(directionVariable);

// Printing data to text file for analysis
  output.println(millis()-startTime + "," + xError + "," + (int)steeringAngle + ","
  + roadWidth + "," + roadCenter);
  prevTime = currentTime;
  prevXError = xError;

// The following statements provide visual feedback for boundary identification
  for ( int i = 1; i < count1; i++)
  {
    line(columnArrayYellow[i-1],rowArrayYellow[i-1],columnArrayYellow[i],rowArrayYellow[i]);
    stroke(50);
    strokeWeight(10);
  }

  for ( int i = 1; i < count2; i++)
  {
    line(columnArrayRed[i-1],rowArrayRed[i-1],columnArrayRed[i],rowArrayRed[i]);
    stroke(50);
    strokeWeight(10);
  }

  noFill();
  ellipse(roadCenter, yCenter, 30, 30);
  fill(250);
  ellipse(xCenter, yCenter, 10, 10);

  count1 = 0;
  count2 = 0;
```

```
}


//The following funcitons came as part of the KinectPV2 library and are used to draw lines
//between joints once a body is being tracked.

void drawBody(KJoint[] joints)
{
  drawBone(joints, KinectPV2.JointType_Head, KinectPV2.JointType_Neck);
  drawBone(joints, KinectPV2.JointType_Neck, KinectPV2.JointType_SpineShoulder);
  drawBone(joints, KinectPV2.JointType_SpineShoulder, KinectPV2.JointType_SpineMid);
  drawBone(joints, KinectPV2.JointType_SpineMid, KinectPV2.JointType_SpineBase);
  drawBone(joints, KinectPV2.JointType_SpineShoulder, KinectPV2.JointType_ShoulderRight);
  drawBone(joints, KinectPV2.JointType_SpineShoulder, KinectPV2.JointType_ShoulderLeft);
  drawBone(joints, KinectPV2.JointType_SpineBase, KinectPV2.JointType_HipRight);
  drawBone(joints, KinectPV2.JointType_SpineBase, KinectPV2.JointType_HipLeft);

  // Right Arm
  drawBone(joints, KinectPV2.JointType_ShoulderRight, KinectPV2.JointType_ElbowRight);
  drawBone(joints, KinectPV2.JointType_ElbowRight, KinectPV2.JointType_WristRight);
  drawBone(joints, KinectPV2.JointType_WristRight, KinectPV2.JointType_HandRight);
  drawBone(joints, KinectPV2.JointType_HandRight, KinectPV2.JointType_HandTipRight);
  drawBone(joints, KinectPV2.JointType_WristRight, KinectPV2.JointType_ThumbRight);

  // Left Arm
  drawBone(joints, KinectPV2.JointType_ShoulderLeft, KinectPV2.JointType_ElbowLeft);
  drawBone(joints, KinectPV2.JointType_ElbowLeft, KinectPV2.JointType_WristLeft);
  drawBone(joints, KinectPV2.JointType_WristLeft, KinectPV2.JointType_HandLeft);
  drawBone(joints, KinectPV2.JointType_HandLeft, KinectPV2.JointType_HandTipLeft);
  drawBone(joints, KinectPV2.JointType_WristLeft, KinectPV2.JointType_ThumbLeft);

  // Right Leg
  drawBone(joints, KinectPV2.JointType_HipRight, KinectPV2.JointType_KneeRight);
  drawBone(joints, KinectPV2.JointType_KneeRight, KinectPV2.JointType_AnkleRight);
  drawBone(joints, KinectPV2.JointType_AnkleRight, KinectPV2.JointType_FootRight);

  // Left Leg
  drawBone(joints, KinectPV2.JointType_HipLeft, KinectPV2.JointType_KneeLeft);
  drawBone(joints, KinectPV2.JointType_KneeLeft, KinectPV2.JointType_AnkleLeft);
  drawBone(joints, KinectPV2.JointType_AnkleLeft, KinectPV2.JointType_FootLeft);

  drawJoint(joints, KinectPV2.JointType_HandTipLeft);
  drawJoint(joints, KinectPV2.JointType_HandTipRight);
  drawJoint(joints, KinectPV2.JointType_FootLeft);
  drawJoint(joints, KinectPV2.JointType_FootRight);

  drawJoint(joints, KinectPV2.JointType_ThumbLeft);
  drawJoint(joints, KinectPV2.JointType_ThumbRight);

  drawJoint(joints, KinectPV2.JointType_Head);
}

//draw joint
void drawJoint(KJoint[] joints, int jointType) {
  pushMatrix();
  translate(joints[jointType].getX(), joints[jointType].getY(), joints[jointType].getZ());
  ellipse(0, 0, 25, 25);
  popMatrix();
}
```

```
//draw bone
void drawBone(KJoint[] joints, int jointType1, int jointType2) {
  pushMatrix();
  translate(joints[jointType1].getX(), joints[jointType1].getY(), joints[jointType1].getZ());
  ellipse(0, 0, 25, 25);
  popMatrix();
  line(joints[jointType1].getX(), joints[jointType1].getY(), joints[jointType1].getZ(),
  joints[jointType2].getX(), joints[jointType2].getY(), joints[jointType2].getZ());
}

//draw hand state
void drawHandState(KJoint joint)
{
  noStroke();
  handState(joint.getState());
  pushMatrix();
  translate(joint.getX(), joint.getY(), joint.getZ());
  ellipse(0, 0, 70, 70);
  popMatrix();
}

void handState(int handState)
{
  switch(handState)
  {
    case KinectPV2.HandState_Open:
      fill(0, 255, 0);
    break;
    case KinectPV2.HandState_Closed:
      fill(255, 0, 0);
    break;
    case KinectPV2.HandState_Lasso:
      fill(0, 0, 255);
    break;
    case KinectPV2.HandState_NotTracked:
      fill(255, 255, 255);
    break;
  }
}
```

A.2 Arduino Code

```
/* This code is designed to control the steering and drivePWM of a 1/4th scale
 *  electric rc car. The steering angle and drivePWM are received through the usb Serial port
 *  from the program called Processing. The steering angle is sent to the servos
 *  controlling the front wheels using PWM with the arduino servo library. The drivePWM
 *  is sent to a Beaglebone Black using the arduino ROS library. drivePWM control is performed
 *  using ROS on the Beaglebone in conjunction with an ELMO Gold solo whistle motor
 *  controller.
 *  Written by Forrest Berg 2-1-16
*/
#include <Servo.h>

Servo steering;
Servo drive;
double angle = 0; // output
double drivePWM=0; //float
int estop = 0;
int directionVariable = 3;
int startCheck = 0;
double centerAngle = 90;
double steeringAngle;
double currentTime = 0;
double prevTime = 0;
double startTime = 0;
double prevPosError = 0;
double errorSum = 0;
double posError;
double currentVel = 0;
double velError;
double prevVelError;
double desiredVel;

int throttleNeutral = 1450;
int minVelocity = 1420;   // lowest possible speed
int maxVelocity = 1350;
int startVelocity = 1380; // startup speed to overcome friction



// Steering Controller Gains
double Kp1=.3, Ki1=0, Kd1=100;


void setup()
{
   Serial.begin(9600); // Start serial communication at 9600 bps
   attachInterrupt(digitalPinToInterrupt(3), eStop, RISING);
   drive.attach(8);
   steering.attach(9);
   steering.write(90);
   drive.writeMicroseconds(1500);
   delay(4000);
}

// The main loop is where the drive motor velocity and steering angles
// are calculated and sent to the motor.

void loop() {
   while (Serial.available() == 0)
   {
    // Do nothing while waiting for data to be sent from processing
```

```
}
posError = Serial.read(); // receiving absolute value of error.
while (Serial.available() == 0)
{
 // Do nothing while waiting for data to be sent from processing
}
directionVariable = Serial.read(); // receiving variable that indicates if
//  the error is positive or negative

drivePWM = minVelocity;

switch(directionVariable)
{
   case 1:
     posError = posError*-1;  // negating position error
     if (startCheck == 0 & directionVariable != 3)
     {
       drive.write(startVelocity);
       delay(200);
       startCheck = 1;
     }
   break;
   case 2:
     // error stays positive in this case.
     if (startCheck == 0 & directionVariable != 3)
     {
       drive.write(startVelocity);
       delay(200);
       startCheck = 1;
     }
   break;
   case 3:
     // when directionVariable = 3 a stop is required
     drivePWM = throttleNeutral;
     startCheck = 0;
   break;
   default:
     // if default case is reached, stop the car
     drivePWM = throttleNeutral;
   break;
}

currentTime = millis();
double dTime = currentTime - prevTime;
double pos_dError = (posError - prevPosError)/dTime;
double vel_dError = (velError - prevVelError)/dTime;
angle = Kp1*posError + Kd1*pos_dError;

// the following statements constrain the angle values to the vehicle's operating range
if (angle < -90) angle = -90; // maximum right turn
if (angle > 90) angle = 90;   // maximum left turn
steeringAngle = 90 + angle;

steering.write(steeringAngle);
prevPosError = posError;
prevVelError = velError;
prevTime = currentTime;

if (estop == 1) drivePWM = throttleNeutral;
drive.write(drivePWM);
```

```
121    delay(1);
122 }
123
124 // Interrupt service routine triggered by manual emergency stop switch
125 void eStop()
126 {
127   estop = 1;
128 }
```